Università di Pisa
Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Magistrale in Informatica

Algoritmi e strutture di dati per la rappresentazione compatta di cammini minimi in grafi

Candidata
Chiara Marcheschi

Relatore
Prof. Roberto Grossi
Controrelatrice
Prof.ssa Maria Grazia Scutellà

Anno Accademico 2012/2013

Indice

Elenco delle figure

Elenco degli algoritmi

Capitolo 1

Introduzione

La seguente tesi rappresenta il documento finale, redatto a conclusione di una prova finale di Laurea Magistrale, riguardante lo studio di algoritmi e strutture di dati per la rappresentazione di *tutti* i cammini minimi[1], presenti in grafi non pesati e non diretti.

È importante osservare che, potenzialmente, per ogni coppia di nodi (s, t) appartenente ad un grafo G, possono esistere un numero esponenziale di cammini minimi, tutti di uguale costo.

I principali (e più comuni) algoritmi di individuazione di cammini minimi, data una *path query*, recuperano un solo cammino minimo per ogni coppia di nodi (s, t), sorgente-destinazione, indicata nella query.

Al contrario, l'obiettivo che ci prefiggiamo è quello di ricercare rappresentare e, su richiesta, recuperare tutti i cammini minimi esistenti in un grafo non pesato e non diretto. Ricerchiamo quindi una soluzione, al problema di rappresentare di tutti i cammini minimi di un grafo, che risulti efficiente in termini di spazio occupato e di tempo impiegato al recupero di tali cammini. Il nostro obiettivo è quello di riuscire a sviluppare una struttura di dati, che permetta di recuperare i cammini minimi di un grafo G (per una data coppia di vertici $(s, t) \in G$) con un costo computazionale, proporzionale al loro numero e alla loro lunghezza. Allo stesso tempo però, è necessario che tale struttura di dati, costruita effettuando (durante una fase di preprocessing) elaborazioni sul grafo G, non costi eccessivamente in termini di spazio. La possibile dimensione esponenziale dell'output (rispetto al numero di vertici n del grafo) rende difficile l'individuazione di una soluzione efficiente in spazio e tempo. Per questo motivo, nel quarto capitolo, analizziamo diverse soluzioni, che risolvono il problema, soffermandoci sui tradeoff esistenti tra le dimensioni delle strutture di dati utilizzate per mantenere le informazio-

[1]Notare che, nei grafi non pesati, i cammini minimi equivalgono ai cammini più brevi (vedi capitolo 2)

ni e il tempo necessario a recuperale. Tutto questo al fine di individuare il gap esistente tra costo in spazio delle strutture compresse e relativo costo in tempo di decompressione delle stesse, in modo da poter sviluppare una soluzione tollerabile, sia in termini di occupazione in memoria, che di costo computazionale.

I capitoli seguenti descrivono il problema da risolvere e, in modo incrementale, arrivano ad una dettagliata descrizione della soluzione individuata. Nel secondo capitolo sono riportati i concetti base del dominio del problema e le strutture di dati, utilizzate per lo sviluppo della soluzione realizzata.

Nel successivo capitolo (capitolo 3) è introdotto il concetto di "ridondanza", concetto tipico della teoria dell'informazione, ma qui applicato ai cammini minimi di un grafo, al fine di evidenziare le similarità esistenti tra di essi. Nello stesso capitolo, sono riportati i risultati dello studio effettuato sull'insieme dei cammini minimi di un grafo G, attraverso l'identificazione di due proprietà: la prima valida per grafi generici, mentre la seconda per grafi non pesati. Tali proprietà rivelano le relazioni esistenti tra la comprimibilità dell'insieme dei cammini minimi e gli elementi strutturali del grafo (archi e nodi).

Individuate e formalizzate alcune delle cause della "ridondanza", nel quarto capitolo, riportiamo alcune possibili soluzioni al problema del recupero di tutti i cammini minimi, analizzandone il costo in termini di tempo e spazio. Le soluzioni considerate servono ad avere una visione globale del problema e della sua complessità.

Le osservazioni effettuate al quarto capitolo sono essenziali per individuare il nostro metodo di risoluzione alternativo, esposto nel capitolo cinque. Alla base della soluzione individuata, vi è una strategia basata sul "divide et impera", che scompone la struttura grafo e il relativo insieme dei cammini minimi. La strategia di scomposizione fa sì che gli elementi, che condividono più informazione, siano mantenuti nella stessa partizione, in modo da poter sfruttare le loro similarità, al fine di comprimere. In questo modo, riusciamo a scomporre il problema in sottoproblemi di dimensione minore, risolverli e ricomporre le diverse parti nella soluzione finale.

Il capitolo cinque prosegue formalizzando la soluzione, da noi sviluppata, attraverso la descrizione delle tre parti che la definiscono: delle strutture ausiliarie, utilizzate per mantenere le informazioni necessarie ad individuare i cammini minimi; della fase di preprocessing, durante la quale si creano tali strutture; e dell'algoritmo di recupero dei cammini minimi, effettuato a partire dalle strutture costruite.

Durante la fase di preprocessing, sono create le strutture ausiliare utilizzate dall'algoritmo di recupero dei cammini minimi. Di seguito, specifichiamo meglio tale processo.

Il grafo, oggetto del recupero dei cammini minimi, è scomposto in più grafi diretti aciclici (DAG) tra loro annidati che, al momento della *path query*, vengono ricorsivamente visitati, al fine di recuperare tutti i cammini minimi esistenti tra una data coppia di nodi sorgente-destinazione. La navigazione dei DAG è effettuata utilizzando due strutture, anch'esse costruite durante la fase di preprocessing, costituite da una tabella di routing e da un insieme di filtri di Bloom. Tali strutture mantengono le informazioni necessarie a visitare solamente i cammini minimi, evitando di "imboccare" cammini di costo non minimo. Così facendo, il problema di recupero di tutti i cammini minimi, per una coppia di nodi (s, t), è ricondotto ad un problema di routing su DAG annidati.

Data la struttura dei DAG e la coppia di vertici (s, t), utilizziamo una tabella di routing, che contiene tutti i nodi di collegamento tra i DAG, detti "nodi ponte", corrispondenti ai nodi da "attraversare" per raggiungere t a partire da s. L'informazione necessaria, per "passare" da un nodo ponte al successivo, è contenuta nei filtri di Bloom, che mantengono le associazioni tra ogni arco dei DAG ed i "nodi ponte" da esso raggiungibili.

Per meglio comprendere gli algoritmi e le strutture di dati sviluppate per la rappresentazione compatta dei cammini minimi, nel capitolo cinque sono anche riportati due esempi di applicazione della soluzione a due diversi grafi. Nello stesso capitolo, è inserito lo pseudocodice delle procedure utilizzate in fase di preprocessing e di recupero dei cammini minimi.

Nel sesto capitolo è riportata l'analisi dei costi della soluzione proposta, costi che risultano essere strettamente legati al numero di ricorsioni necessarie alla costruzione della struttura dei DAG annidati. Per questo motivo, data la non banalità del problema di stimare il numero di ricorsioni, nel capitolo sei, sono presenti anche i risultati delle analisi sperimentali, effettuate per stimare tale valore. Questi esperimenti mostrano che il numero di ricorsioni tende a rimanere "basso". Per questo motivo, l'aumento dei costi, legato al processo ricorsivo, risulta irrilevante rispetto al costo del primo passo di ricorsione, quello effettuato sull'intero grafo.

L'analisi dei costi mostra come il nostro algoritmo di recupero dei cammini minimi, per un grafo G non diretto e non pesato, sia *output sensitive*. Il numero di passi effettuati sulla struttura di dati, creati in fase di preprocessing di G, al fine di recuperare tutti i cammini minimi per una data coppia di vertici del grafo $(s, t) \in G$, risulta proporzionale al numero di tali cammini ed alla loro lunghezza.

Per quanto riguarda la struttura di dati utilizzata, costituita dai DAG annidati, dai filtri di Bloom e dalla tabella di routing, non riusciamo ad effettuare un'esaustiva analisi teorica sulla loro dimensione. In particolare, anche se attuiamo una serie di strategie per limitare la dimensione della

tabella di routing entro $\mathcal{O}(n^2)$, la dimostrazione di tale limite non risulta banale. L'analisi (o la stima tramite uno studio empirico) della dimensione della tabella di routing è demandata a successivi sviluppi della soluzione.

L'ultimo capitolo (capitolo sette) ripercorre lo studio fatto per il problema di rappresentare tutti i cammini minimi di un grafo, fino ad arrivare all'individuazione di una soluzione, e riporta alcune strade di ricerca aperte dal nostro lavoro, oggetto di possibili sviluppi futuri.

Capitolo 2

Grafi, distanze e visite

Nel seguente capitolo andiamo ad illustrare i concetti di base necessari alla comprensione dell'intera tesi, definendo, in modo incrementale, il dominio del problema e gli strumenti utilizzati per lo sviluppo di una soluzione. Andiamo, pertanto, a riportare alcune definizioni[1], comunemente note in ambito matematico, ma qui inserite perché necessarie per la corretta comprensione della terminologia usata nei successivi capitoli e, di conseguenza, essenziali per capire le analisi e i risultati dello studio. Tali definizioni riguardano la struttura di *grafo*, i concetti di *cammino*, di *cammino minimo* e di *distanza*. In seguito riportiamo alcuni algoritmi base per l'individuazione di cammini minimi in un grafo, concludendo con la descrizione di alcune strutture dei dati, necessarie allo sviluppo delle diverse soluzioni illustrate nei successivi capitoli.

2.1 Grafi

Con il termine "grafo" si indica una struttura matematica discreta, usata in svariati campi applicativi. Tale struttura si presta a modellare e rappresentare una grande quantità di situazioni. Questo spiega l'importanza dei grafi nella matematica applicata e in particolare nella ricerca operativa: una grande quantità di problemi può essere espressa per mezzo di problemi su grafi. Di conseguenza, gli algoritmi efficienti sui grafi rappresentano alcuni strumenti generali per la risoluzione di numerosi problemi di rilevanza pratica e teorica.

Definizione 1.0.1. *Un* grafo *non orientato G è una coppia di insiemi finiti $G = (V, E)$, dove V rappresenta un insieme di* nodi *(o vertici), mentre*

[1] Definizioni tratte da [1, 4, 25]

$E \subseteq V \times V$ *un sottoinsieme di coppie di nodi; ogni coppia di nodi prende il nome di* arco.

Ci riferiamo all'insieme dei nodi e degli archi di un grafo G rispettivamente con $V(G) = \{v_1, ..., v_n\}$ e $E(G) = \{e_1, ..., e_m\}$, dove $n = |V|$, mentre $m = |E|$.

Definizione 1.0.2. *La* dimensione *di un grafo è data dalla somma tra* n *e* m*.*

La complessità lineare viene riferita al costo $\mathcal{O}(n+m)$ ed è perciò definita considerando la somma dei due parametri. In generale, vale $0 \leqslant m \leqslant \binom{n}{2}$, poiché il numero massimo di archi è dato dal numero di tutte le possibili coppie di nodi, che sono $\binom{n}{2} = \mathcal{O}(n^2)$. Un grafo è *sparso* se $m = \mathcal{O}(n)$ e *denso* se $m = \Theta(n)$.
Nella trattazione di grafi non orientati, un arco corrisponde ad un collegamento tra due nodi u e v ed è rappresentato con la notazione (u, v), equivalente alla notazione (v, u). Graficamente, tale collegamento corrisponde ad una linea che collega la coppia di nodi u e v, rappresentati come cerchi (figura 2.1).

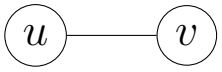


Figura 2.1: Arco non orientato $(u, v) = (v, u)$

Definizione 1.0.3. *Un* grafo diretto *è una coppia di insiemi finiti* $G = (V, E)$*, dove* V *rappresenta l'insieme dei* nodi *(o vertici), mentre* $E \subseteq V \times V$ *un sottoinsieme di coppie* ordinate *di nodi; ogni coppia di nodi prende il nome di* arco *orientato.*

Dato l'arco orientato $e = (u, v)$, il primo nodo u è detto testa, il secondo è detto coda. Si dice che l'arco è uscente da u ed entrante in v.

Nella trattazione di grafi orientati, un arco (u, v) corrisponde ad un collegamento percorribile in un unico verso, tra il nodo u ed il nodo v; per questo non sarà possibile transitare, dal nodo v al nodo u, a meno della presenza di un diverso arco orientato (v, u). Graficamente, tali collegamenti corrispondono ad una freccia che collega il nodo testa al nodo coda, entrambi rappresentati da cerchi (figura 2.2).

Definizione 1.0.4. *Un grafo è detto* pesato *(o etichettato) sugli archi se è definita una funzione* $W : E \mapsto \mathbb{R}$*, che assegna un valore (reale) ad ogni arco del grafo. Il valore associato è detto* peso*, costo o guadagno a seconda della semantica associata ad esso. Un grafo pesato sarà quindi una tripla* $G = (V, E, W)$*.*

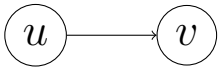


Figura 2.2: Arco orientato (u, v)

Graficamente, il peso di ogni arco verrà riportato in corrispondenza della rappresentazione grafica dell'arco tramite un'etichetta o un valore (reale). Il costo di un arco di un grafo non pesato è, per convenzione, uguale a costo unitario.

Figura 2.3: Due diverse rappresentazioni dell'arco pesato $e = (u, v)$ con peso $W(e) = w = 1.5$

2.2 Cammini e distanze

Definizione 2.0.5. *Un* cammino P *da un nodo* u *ad un nodo* v *è una sequenza di nodi* $v_1, v_2, \ldots, v_k$*, tali che* $v_1 = u$*,* $v_k = v$ *e tali che esiste un arco* $(v_i, v_{i+1}) \in E$ *per ogni* $1 \leqslant i < k$*. L'intero* k *è detto lunghezza del cammino.*

Un *ciclo* è un cammino per cui vale $v_1 = v_k$, ossia un cammino che ritorna nel nodo di partenza. Un cammino è *semplice* se non attraversa alcun nodo più di una volta, ossia se non esiste alcun ciclo annidato al suo interno[2]. Due nodi u e v sono *connessi* se, e solo se, esiste un cammino tra essi. Un grafo in cui ogni coppia di nodi è connessa è detto, a sua volta, connesso. Posta, come ipotesi, l'univocità di ogni arco $e \in E$, la sequenza di nodi, $v_1, \ldots, v_k$, costituenti un cammino P, corrisponde univocamente ad una sequenza di archi $e_1, \ldots, e_{k-1}$, tali che $e_i = (v_i, v_{i+1})$. Assunto ciò, possiamo considerare una definizione alternativa di cammino, di seguito riportata.

Definizione 2.0.6. *Un* cammino P *(di lunghezza* k*), da un nodo* u *ad un nodo* v *è una sequenza di archi,* $e_1, \ldots, e_{k-1}$*, tali che* $e_i \in E$*,* $e_1 = (u, v_2)$ *e* $e_{k-1} = (v_{k-1}, v)$*.*

Definito un cammino, come una sequenza di archi, di seguito riportiamo la definizione di costo di un cammino.

[2]Notare che ogni cammino minimo di un grafo è un cammino semplice.

Definizione 2.0.7. *Il* costo *(o lunghezza)* di un cammino P*, è dato dalla somma dei pesi (costi) associati agli archi che lo costituiscono.*

$$W(P) = \sum_{(u,v)\in P} w(u,v)$$

.

Il costo di un cammino, da u a v, in un grafo non pesato, equivale al numero di archi che costituiscono il cammino, cioè al numero di "hop" (salti) effettuati, transitando da un nodo al successivo del cammino, fino al raggiungimento del nodo v. Definito $\mathcal{P}_G$ l'insieme di tutti i cammini in un grafo G, e $\mathcal{P}_G(u,v)$ l'insieme di tutti i cammini da u a v nello stesso grafo, riportiamo la definizione di cammino minimo.

Definizione 2.0.8. *Un* cammino minimo $\bar{P}$*, da un nodo u ad un nodo v, è un cammino, che ha costo minimo rispetto a tutti i possibili cammini da u a v.*

Segue che, data W funzione di costo,

$$\bar{P} \text{ è minimo per la coppia } (u,v) \Leftrightarrow \nexists P \in \mathcal{P}_G(u,v) : W(P) < W(\bar{P})$$

.

Definizione 2.0.9. *La* distanza *(o metrica) su un insieme X è una qualsiasi funzione $d : X \times X \mapsto \mathbb{R}$, che soddisfa le seguenti proprietà $\forall x, y, z \in X$:*

1. $d(x,y) \geqslant 0$
2. $d(x,y) = 0 \Leftrightarrow x = y \in X$
3. $d(x,y) = d(y,x)$*, proprietà simmetrica*
4. $d(x,y) \leqslant d(x,z) + d(z,y)$*, disuguaglianza triangolare*

Dato un grafo G si può definire una metrica sui nodi considerando la funzione che associa ad ogni coppia (u,v) la lunghezza di un cammino minimo da u a v (notare che, per la proprietà 3 del teorema, tale lunghezza equivale alla lunghezza del cammino minimo da v a u). Se i nodi non sono connessi, la distanza tra di essi è pari a $+\infty$. Da notare che, nel caso in cui G sia un grafo diretto, la proprietà simmetrica del teorema non ha validità; di conseguenza, la metrica definita per G soddisfa solamente tre delle quattro proprietà.

Definizione 2.0.10. *Il* diametro *di un grafo G è la massima distanza tra due vertici.*

Definizione 2.0.11. *Dati due nodi u e $v \in G$, con G grafo non diretto e non pesato, la distanza tra u e v, $d(u,v)$, è uguale al* numero minimo di hop *che intercorrono tra i due nodi.*

2.3 Algoritmi per il recupero di cammini minimi

Path e distance query sono sicuramente due tra i più importanti problemi di ottimizzazione su grafi, la cui rilevanza ha portato al loro studio e allo sviluppo di algoritmi di risoluzione, fin dagli anni '50. È importante notare che, quando si parla di algoritmi di ottimizzazione su grafi, si tende a classificarli in base al problema che trattano: Single Source Shortest Path (SSSP), Multiple Source Shortest Path (MSSP), All Pairs Shortest Path (APSP).

Il Single Source Shortest Path (SSSP) Problem si pone l'obiettivo di calcolare un cammino minimo, a partire da una data sorgente, verso tutti i vertici del grafo; invece, il Multiple Source Shortest Path (MSSP) Problem richiede di trovare un cammino minimo per tutte le coppie sorgente-destinazione, *fissata la destinazione*; infine l'All Pairs Shortest Path (APSP) Problem si propone di recuperare un cammino minimo per ogni coppia di nodi sorgente-destinazione.

Tra i più conosciuti e importanti algoritmi di risoluzione per i problemi descritti si ricordano:

Dijkstra per la risoluzione di SSSP su un grafo con pesi non negativi sugli archi;

Bellman-Ford per la risoluzione di SSSP su un grafo, anche se i pesi sono negativi;

Floyd-Warshall per la risoluzione di APSP su un grafo pesato e orientato.

Il fatto che la struttura del grafo si presti molto a modellare e rappresentare una grande quantità di situazioni, porta, anche oggi, la comunità scientifica e di ricerca ad avere un grande interessamento nei suoi riguardi. La ricerca e lo sviluppo di algoritmi su grafi è ancora molto intensa, ed i risultati ottenuti hanno impiego in svariati campi: dal web, alle reti stradali, a quelle sociali etc.

Tra i più recenti studi per lo sviluppo di algoritmi e strutture di dati sviluppati su grafi, si ricordano, per le loro analisi ed eccellenti risultati, quelli riguardanti la risoluzione di problemi di ottimizzazione su grafi planari di Mozes, Sommer, Weimann, Klein, [18, 22, 17], Fakcharoenphol e Rao [9, 10]. Per una miglior analisi degli algoritmi, riguardanti i cammini minimi, rimandiamo al lavoro di tesi di dottorato di Sommer [25] e ad un ulteriore suo articolo [26], recentemente pubblicato.

Altri due lavori che propongono soluzioni per il problema dell'All Pairs Shortest Path (APSP), interessanti per l'originalità delle osservazioni fatte, sono quelli di Ferragina, Nitto, Venturini [11] e di Demetrescu e Italiano [6].

2.4 Strutture dei dati

Il nostro obiettivo è quello di sviluppare e utilizzare algoritmi e strutture di dati per poter "potenzialmente" recuperare *tutti* i cammini minimi per una *qualsiasi* coppia di vertici di un grafo non pesato e non diretto, (non limitandoci ad un unico cammino per ogni coppia, come nell'MSSP Problem).

Di seguito, descriviamo alcune strutture utilizzate nelle soluzioni descritte nei successivi capitoli.

2.4.1 DAG

Definizione 4.1.1. *Un* Directed Acyclic Graph (DAG) *è un grafo diretto aciclico, ovvero privo di cicli. In un DAG, comunque scegliamo un vertice del grafo, non possiamo trovare un cammino di archi diretti, che inizi e termini in esso.*

Da notare che, dato un DAG e presi due nodi u e v appartenenti ad esso, se esiste un cammino, da u a v, allora non esiste un cammino da v a u.

2.4.2 BDD

Definizione 4.2.1. *Un* Binary Decision Diagram (BDD) *è una struttura di dati usata per rappresentare funzioni Booleane o insiemi e relazioni in modo compresso.*

Un BDD è rappresentato graficamente come un albero binario, i cui nodi corrispondono a variabili, o a elementi, a seconda di ciò che stiamo rappresentando (una funzione booleana o un insieme). Ogni nodo ha due archi uscenti, rappresentati con una linea tratteggiata e una continua, che indicano, rispettivamente, l'assegnamento alla variabile del valore $falso$ o del valore *vero* (nel caso di rappresentazione di insiemi, indicano l'assenza o la presenza nell'insieme dello specifico elemento, corrispondente al nodo). Se interpretiamo un BDD, come la rappresentazione di una funzione, ogni cammino da "radice" a "foglia" implica la scelta di determinati valori booleani per le variabili incontrate; mentre, se un BDD rappresenta un insieme, ogni cammino specifica la presenza, o l'assenza, degli elementi corrispondenti ai

diversi nodi. In un BDD ogni cammino termina in una foglia, di norma rappresentata in forma rettangolare. Nel caso delle funzioni, tale foglia indica il valore booleano della funzione corrispondente allo specifico assegnamento definito dal cammino, mentre, nel caso delle rappresentazioni di insiemi, serve a classificare gli elementi di un insieme, sotto due classi distinte.

Il principale vantaggio di rappresentare gli insiemi in modo compresso, tramite le BDD, è il fatto che è possibile operare con gli operatori insiemistici direttamente sulle strutture compresse.

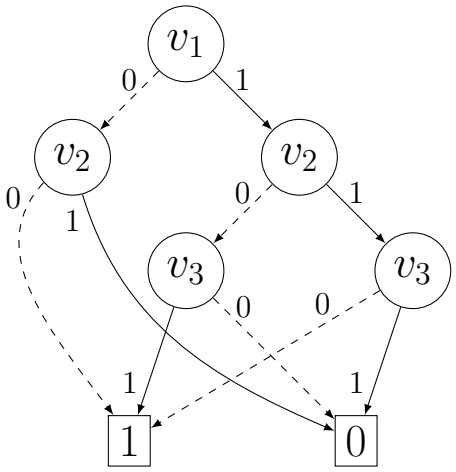


Figura 2.4: BDD su tre elementi che definisce la funzione booleana $f(v_1, v_2, v_3)$
$f(0, 0, v_3) = 1$, $f(0, 1, v_3) = 0$ ramo sinistro di v_1
$f(1, 0, 1) = 1$, $f(1, 0, 0) = 0$, $f(1, 1, 0) = 1$, $f(1, 1, 1) = 0$ ramo destro

L'uso delle BDD nella rappresentazione di grafi

Come precedentemente osservato, i cammini possono essere definiti come sequenze di nodi o sequenze di archi, ma, mentre nella prima definizione l'ordinamento degli elementi (nodi) è un'informazione essenziale, nella seconda definizione l'ordinamento degli elementi (archi) è deducibile dagli elementi stessi. Per questo, posta l'univocità degli archi (definiti come coppie di nodi), si fa notare che, dato un qualsiasi cammino P, se si memorizza come insieme di archi (cioè se ne ignora l'ordinamento relativo dato dall'ordine in cui compaiono in P), è sempre e comunque possibile recuperare il cammino P di partenza, attraverso un semplice ordinamento degli archi. Questo principio è alla base della rappresentazione dei cammini tramite l'uso di BDD; infatti, tramite queste strutture di dati, è possibile rappresentare in modo compresso insiemi di cammini definiti come insiemi di archi, ignorandone l'ordine. Per maggiori informazioni riguardo i BDD e le altre strutture, da

essi derivate (ZDD, SBDD ...), consultare [20]. Per studiare nel dettaglio l'applicazione dei BDD per la rappresentazione di cammini minimi in un grafo, si veda [20, 19, 28, 14, 15, 27, 28, 30].

Un altro lavoro che combina BDD e grafi (con minor efficacia in termini di compressione), attraverso un diverso approccio di memorizzazione, si trova nell'articolo di Dong e Molitor [7].

2.4.3 Filtri di Bloom

Un filtro di Bloom, più comunemente chiamato *Bloom filter*, è una struttura di dati probabilistica, sviluppata da Burton Howard Bloom nell'1970 [2], che viene usata per testare se un dato elemento è, o meno, presente in un insieme.

Definizione 4.3.1. *Un* Bloom filter *è una struttura di dati probabilistica, usata per rappresentare insiemi, che consiste in un array di M bit e di k funzioni hash, ognuna delle quali prende un elemento dell'insieme e lo mappa su una cella dell'array.*

Operazioni effuettuabili su di un Bloom filter:

inserimento di un elemento: per aggiungere un elemento all'insieme rappresentato dal Bloom filter, applichiamo le k funzioni hash all'elemento e "settiamo" ad 1 le posizioni dell'aray risultanti dalle applicazioni;

verifica di presenza: per verificare se un elemento è presente o meno, nell'insieme rappresentato dal bloom filter, applichiamo le k funzioni hash all'elemento e mettiamo in *and* i bit contenuti nelle celle corrispondenti ai valori risultanti dalle applicazioni delle funzioni. Se, così facendo, otteniamo un valore pari a 0, significa che l'elemento non è presente; al contrario, se il valore risultante è 1, l'elemento è presente con una certa probabilità, dipendente da M, k e dalla cardinalità N dell'insieme inserito nel Bloom filter.

Probabilità di avere falsi positivi

Assunto che, per ogni funzione hash h, tutti i valori (da 0 a $M-1$) abbiano la stessa probabilità di essere restituiti da h, dopo l'inserimento di N elementi, la probabilità che un certo bit sia settato ad 1 è:

$$1 - \left(1 - \frac{1}{M}\right)^{kN}$$

La probabilità che k bit siano settati ad 1 equivale a:

$$p = \left[1 - \left(1 - \frac{1}{M}\right)^{kN}\right]^k \approx \left(1 - e^{-\frac{kN}{M}}\right)^k$$

Tale probabilità corrisponde alla probabilità di avere un falso positivo. Per maggiori chiarimenti si veda il lavoro di Broder e Mitzenmacher [3].

Dimensione dei filtri di Bloom

Fissati i valori della probabilità desiderata p (probabilità di avere un falso positivo durante un'interrogazione) e della cardinalità N dell'insieme degli elementi inseriti nel Bloom filter, è possibile determinare la dimensione (in bit) M del filtro.

$$M = -\frac{N \ln p}{(\ln 2)^2} \; bit$$

Per ripercorrere l'analisi, che porta a tale valore, si veda il lavoro di Broder e Mitzenmacher [3].

2.5 Visite in un grafo

La visita di un grafo è l'esplorazione dei vertici del grafo, effettuata a partire da un certo nodo, seguendo gli archi per passare da un nodo al successivo. In particolare, riportiamo la definione di visita in ampiezza, perché essa è alla base degli algoritmi di costruzione da noi sviluppati, descritti nel capitolo 5.

Definizione 5.0.2. *La* visita in ampiezza*, o breadth-first-search (BFS), di un grafo G, dato un vertice sorgente s, consiste nell'esplorazione sistematica di tutti i vertici raggiungibili da s, in modo tale da esplorare tutti i vertici che hanno distanza k, prima di iniziare a scoprire quelli che hanno distanza $k+1$, con $1 \leq k < d$, detta d la massima distanza da s di un nodo $v \in G$*

A pagina seguente, riportiamo l'algoritmo che implementa una visita BFS su di un grafo G, a partire da un vertice s. La visita è effettuata con l'aiuto di una coda Q (FIFO) nella quale vengono sistematicamente inseriti i vertici non ancora incontrati. Nell'algoritmo, per determinare se un nodo sia stato già visitato, o meno, marchiamo i nodi man mano che li incontriamo (istruzione $M.add(c)$). Il primo nodo, ad essere inserito in coda, è il nodo s. L'algoritmo estrae un nodo alla volta dalla coda ($Q.push(s)$), inserendone i vicini (se visti per la prima volta) all'interno della coda stessa.

Algoritmo 1 Visita BFS, generazione dei nodi in ordine BFS rispetto al nodo s

```
procedure BFS(s)
    Q ← new_queue()                          ▷ Coda visita BFS
    Q.push(s)
    M = {s}                                  ▷ Insieme dei nodi visitati
    while Q is not empty do
        v ← Q.pop()
        yield v
        for all c ∈ v.neighbors() do
            if c is not in M then
                M.add(c)
                Q.push(c)
            end if
        end for
    end while
end procedure
```

Capitolo 3

"Ridondanza" d'informazione nei cammini minimi

Nel seguente capitolo, dopo aver definito il problema che ci prefiggiamo di risolvere, analizzato le soluzioni attualmente esistenti ed aver osservato che esiste ancora spazio di ricerca e di miglioramento, andiamo ad esporre ed analizzare la comprimibilità dell'insieme dei cammini minimi. Tale comprimibilità deriva dalle strutture di connessione dei grafi: la presenza (o l'assenza) di un arco, che colleghi due vertici, può fortemente influenzare l'insieme dei cammini e le loro similarità. Dopo aver evidenziato le similarità tra i cammini minimi di un grafo, ne ricerchiamo le cause, arrivando ad individuarne due, successivamente specificate nelle proprietà 3.1 e 3.2.

Problema e stato dell'arte

Il problema, per il quale ricerchiamo una soluzione, consiste nell'individuazione e nel mantenimento di tutti i possibili cammini minimi esistenti tra tutte le coppie di nodi, in modo da poter recuperare tutti i cammini minimi di un grafo G non diretto e non pesato, per una specifica coppia di nodi.

Lo scopo della ricerca è quello di trovare una strategia di rappresentazione dei suddetti cammini, individuando una soluzione che trovi il giusto compromesso tra la memoria occupata dalla struttura di rappresentazione dei cammini minimi ed il tempo impiegato dalla generazione degli stessi, per una data coppia di vertici. L'analisi del problema posto è particolarmente interessante, poiché non è facile trovare trattazioni scientifiche a riguardo. In questo lavoro di tesi, per tale problema, descriviamo diverse soluzioni (capitolo 4), focalizzandoci su una in particolare (capitolo 5), risultante da tutte le osservazioni fatte durante lo studio.

L'oggetto dello studio è originale perché, invece di considerare un solo cammino minimo per ogni coppia di nodi (u, v), analizziamo l'insieme dei cammini minimi di un grafo, nella sua totalità; possono, infatti, esistere un numero esponenziale di cammini minimi per (u, v), tutti di ugual costo.

Lo studio effettuato, e qui descritto, consiste nel ricercare proprietà e caratteristiche strutturali per *l'intero* insieme dei cammini minimi, così da poterli rappresentare in modo compatto. Si cerca perciò di sviluppare un algoritmo su cammini minimi che esca dalla classificazione precedentemente indicata (v. par. 2.3), poiché risolve un problema di ordine superiore, in quanto il risultato dell'algoritmo "contiene" i risultati dei problemi precedentemente elencati (Single Source Shortest Path (SSSP), Multiple Source Shortest Path (MSSP) e All Pairs Shortest Path (APSP)).

È importante annotare che, oltre ai più classici problemi, esiste un problema, anch'esso ampiamente studiato, che ne è la generalizzazione, denominato "k-shortest paths problem". Tale problema si occupa di recuperare k cammini in ordine di costo, per una data coppia di vertici (s, t) appartenente ad un dato grafo G.

Un algoritmo di risoluzione, per il "k-shortest paths problem" in grafi diretti pesati (permettendo anche il recupero di cammini non semplici), è conosciuto dal 1975 (articolo di Fox [12]) ed ha un costo di $\mathcal{O}(m + kn \log n)$. Nel 1998 Eppstein [8] è riuscito a migliorare il precedente risultato, utilizzando una struttura che rappresenta implicitamente i cammini. Il costo di costruzione di tale struttura è pari a $\mathcal{O}(m + n \log n + kn)$, mentre il recupero di ogni cammino (dei k da recuperare) costa $\mathcal{O}(n)$.

Il problema di determinare k cammini minimi semplici (cioè privi di cicli), per grafi diretti pesati, conosce la sua attuale migliore soluzione dal 1971, sviluppata da Yen [29] e generalizzata da Lawler [21]. La soluzione di Yen, implementata con le recenti strutture dati, al caso pessimo, costa $\mathcal{O}(kn(m + n \log n))$, mentre al caso medio, calcola ogni singolo cammino in $\mathcal{O}(n)$. Successivamente, Gotthilf e Lewenstein hanno migliorato tale limite portando il costo a $\mathcal{O}(kn(m + n \log \log n))$ [13].

Per quanto riguarda la risoluzione del "k-shortest paths problem" in grafi non diretti, attualmente, l'algoritmo di risoluzione migliore è di Katoh, Ibaraki e Mine [23] e costa $\mathcal{O}(k(m + n \log n))$. Recentemente, per il recupero dei k cammini minimi è stato sviluppato un algoritmo (per grafi diretti) randomizzato che costa $\tilde{\mathcal{O}}(km\sqrt{n})$ (Roditty e Zwick [24]).

Il problema risolto dall'algoritmo di Katoh, è simile a quello che ci siamo posti. Infatti, un qualsiasi algoritmo che individui, in un grafo non diretto e non pesato, i k cammini di costo minimo, per una coppia di vertici (s, t), può essere utilizzato, con un valore elevato di k, al fine di individuare tutti i cammini minimi esistenti tra s e t. Nonostante questo, poiché l'algoritmo

di Katoh (come quello di Roditty) ha un costo indipendente dal numero di cammini recuperati e dalla loro lunghezza, ha senso studiare il problema di individuare tutti i cammini di costo minimo, al fine di ottenere una soluzione "output sensitive". Utilizzando gli algoritmi sopra esposti paghiamo, per ogni cammino recuperato, rispettivamente $\mathcal{O}(m + n \log n)$ e $\tilde{\mathcal{O}}(m\sqrt{n})$[1], costo molto lontano dall'obiettivo preposto: per ogni cammino pagare in modo proporzionale alla sua lunghezza.

Le analisi, qui riportate, si prefiggono di scoprire ed esplicitare gli elementi essenziali per poter sfruttare le "similarità" esistenti tra gli elementi dell'insieme dei cammini minimi, così da poter creare una struttura che mantenga i dati compressi e, allo stesso tempo, sia di facile decompressione[2]. Osservare l'insieme dei cammini minimi nel suo complesso permette di individuare le similarità tra i diversi elementi; ed è proprio guardando il problema da un più alto livello, che riteniamo possibile trovare gli elementi chiave per la sua risoluzione.

Proprietà dei cammini minimi

L'osservazione di base da cui è scaturito il lavoro di ricerca è la seguente: se esprimiamo in modo esplicito, come sequenze di nodi, tutti i cammini minimi di un grafo, noteremo una forte "ridondanza".

Per poter meglio esprimere e visualizzare il concetto di "ridondanza" su cammini minimi, definiamo due funzioni: una funzione base, che mette in relazione ogni nodo $v \in V$ con un'etichetta univoca identificativa, scelta in un alfabeto Λ (3.1), e una seconda funzione parziale (derivata dalla prima), che mette in corrispondenza biunivoca l'insieme dei cammini esistenti in un grafo, con un sottoinsieme dell'insieme delle stringhe, cioè di sequenze di caratteri (3.2). Questa seconda funzione esprime qualsiasi cammino (definito come sequenza di nodi), attraverso una stringa generata concatenando le etichette dei nodi del cammino, nell'ordine di attraversamento dal nodo sorgente al nodo terminazione.

Definite tali funzioni, possiamo mappare ogni cammino $P \in \mathcal{P}_G$ su una specifica stringa $p \in 2^{\Lambda}$, per meglio analizzare i sottocammini di P considerando le sottostringhe di p. Procedendo nell'analisi, possiamo estendere

[1] L'algoritmo di Katoh può essere esteso ai grafi non pesati, considerando il peso di ogni arco pari ad 1, mentre quello di Roditty può operare su grafi non diretti, se trasformati in grafi diretti.

[2] Trattasi della struttura dei DAG annidati, ampiamente descritta al capitolo 5

$$l : V \mapsto \Lambda, \tag{3.1}$$
$$s : \mathcal{P}_G \mapsto 2^{\Lambda} \tag{3.2}$$

Funzioni 1: Funzioni che mappano rispettivamente nodi su caratteri 3.1 e cammini su stringhe 3.2

e applicare concetti tipici delle stringhe al dominio dei cammini di un grafo, concentrandoci sulla compressione delle informazioni, piuttosto che sulla struttura del grafo.

Il concetto di "ridondanza" è un concetto tipico della teoria dell'informazione, quindi applicabile alle stringhe, in quanto sequenze di simboli generati da una sorgente. Per poter ragionare, a livello qualitativo, sulla comprimibilità dell'insieme dei cammini minimi, vi applichiamo la funzione 3.2 e osserviamo la ridondanza presente nelle stringhe risultanti. In questo modo riusciamo ad estendere il concetto di ridondanza ai cammini[3].

Di seguito, riportiamo un esempio che mostra la forte ridondanza presente nell'insieme dei cammini minimi di un grafo. Consideriamo il grafo in figura 3.1 e riportiamo in tabella (tab. 3.1 e 3.2) tutti i cammini da v_1 a v_{12} e i relativi sottocammini.

Attraverso il suddetto esempio, possiamo notare che *l'insieme dei sottocammini dei cammini minimi da v_x a v_y* (cammini generati tramite una visita BFS effettuata a partire da v_x) *contiene tutti e soli i cammini minimi da v_i a v_j, con v_i e v_j nodi attraversati percorrendo un cammino minimo da v_x a v_y.*

Ogni riga delle (quattro numerate) considera un diverso cammino minimo da v_1 a v_{12} e tutti i relativi sottocammini in esso "contenuti". Prendendo, ad esempio, la prima riga delle due tabelle osserviamo che il cammino minimo $v_1v_2v_6v_9v_{12}$ ha come sottocammini $v_1v_2v_6v_9$, $v_2v_6v_9v_{12}$, $v_1v_2v_6$, $v_2v_6v_9$, $v_6v_9v_{12}$, v_1v_2, v_2v_6, v_6v_9, v_9v_{12}, che sono rispettivamente cammini minimi per le coppie di vertici (v_1, v_9), (v_2, v_{12}), (v_1, v_6), (v_2, v_9), (v_6, v_{12}), (v_1, v_2), (v_2, v_6), (v_6, v_9) e (v_9, v_{12}).

Nella tabella 3.3 inseriamo i cammini minimi da v_5 a v_{12}, gli unici cammini minimi terminanti in v_{12} che non sono sottocammini di cammini minimi da v_1 a v_{12}.

[3]In teoria dell'informazione, la ridondanza è definita come $R = 1 - \frac{H}{H_{max}}$, dove H è l'entropia della sorgente e H_{max} è l'entropia massima (di una sorgente senza memoria e con simboli equiprobabili)

Notare che, anche se tali cammini non sono sottocammini dei cammini minimi da v_1 a v_{12}, poiché non esiste un cammino minimo da v_1 a v_{12} passante per v_5, i cammini minimi da v_5 a v_{12} condividono parte del cammino con i cammini minimi da v_1 a v_{12} : i cammini $v_1v_2v_6v_9v_{12}$ e $v_5v_6v_9v_{12}$ contengono entrambi la sottostringa $v_6v_9v_{12}$, mentre $v_1v_2v_7v_{10}v_{12}$ e $v_5v_7v_{10}v_{12}$ hanno in comune la stringa $v_7v_{10}v_{12}$.

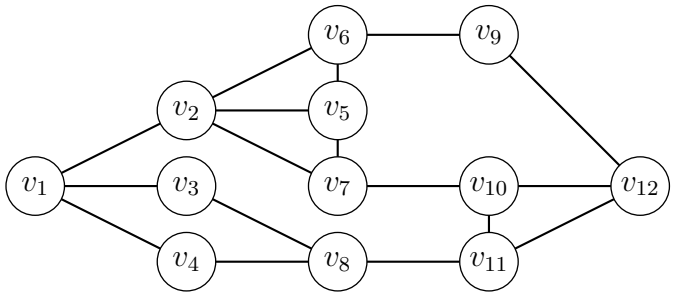


Figura 3.1: Grafo campione

(1)	$\boldsymbol{v_1v_2v_6\ v_9\ v_{12}}$	$v_1v_2v_6\ v_9$	$v_2v_6\ v_9\ v_{12}$	$v_1v_2v_6$	$v_2v_6\ v_9$	$v_6\ v_9\ v_{12}$
(2)	$\boldsymbol{v_1v_2v_7v_{10}v_{12}}$	$v_1v_2v_7v_{10}$	$v_2v_7v_{10}v_{12}$	$v_1v_2v_7$	$v_2v_7v_{10}$	$v_7v_{10}v_{12}$
(3)	$\boldsymbol{v_1v_3v_8v_{11}v_{12}}$	$v_1v_3v_8v_{11}$	$v_3v_8v_{11}v_{12}$	$v_1v_3v_8$	$v_3v_8v_{11}$	$v_8v_{11}v_{12}$
(4)	$\boldsymbol{v_1v_4v_8v_{11}v_{12}}$	$v_1v_4v_8v_{11}$	$v_4v_8v_{11}v_{12}$	$v_1v_4v_8$	$v_4v_8v_{11}$	$v_8v_{11}v_{12}$

Tabella 3.1: Tutti i cammini minimi da v_1 a v_{12} (scritti in **grassetto**), seguiti da alcuni dei loro sottocammini

(1)	$v_1v_2v_6$	$v_2v_6\ v_9$	$v_6\ v_9\ v_{12}$	v_1v_2	v_2v_6	$v_6\ v_9$	$v_9\ v_{12}$
(2)	$v_1v_2v_7$	$v_2v_7v_{10}$	$v_7v_{10}v_{12}$	v_1v_2	v_2v_7	v_7v_{10}	$v_{10}v_{12}$
(3)	$v_1v_3v_8$	$v_3v_8v_{11}$	$v_8v_{11}v_{12}$	v_1v_3	v_3v_8	v_8v_{11}	$v_{11}v_{12}$
(4)	$v_1v_4v_8$	$v_4v_8v_{11}$	$v_8v_{11}v_{12}$	v_1v_4	v_4v_8	v_8v_{11}	$v_{11}v_{12}$

Tabella 3.2: I rimanenti (rispetto alla tabella 3.1) sottocammini per i cammini precedentemente scritti in grassetto

(5)	$\boldsymbol{v_5}$,$\boldsymbol{v_7}$,$\boldsymbol{v_{10}}$,$\boldsymbol{v_{12}}$	$v_5v_7v_{10}$	$v_7v_{10}v_{12}$	v_5v_7	v_7v_{10} ; $v_{10}v_{12}$
(6)	$\boldsymbol{v_5}$,$\boldsymbol{v_6}$, $\boldsymbol{v_9}$, $\boldsymbol{v_{12}}$	$v_5v_6\,v_9$	$v_6\,v_9\,v_{12}$	v_5v_6	$v_6\,v_9$; $v_9\,v_{12}$

Tabella 3.3: Cammini minimi da v_5 a v_{12} (in **grassetto**), con relativi sottocammini

La problematica principale nel voler mantenere tutti i cammini minimi esistenti nel grafo è di tipo spaziale (vedi capitolo 4): la quantità di spazio necessaria a memorizzare tutti i possibili cammini minimi di un generico grafo è esponenziale nel numero dei nodi. Dalle osservazioni precedenti, si può però, facilmente osservare che rappresentare tutti i cammini minimi di un grafo in maniera esplicita porta alla memorizzazione di un'enorme quantità di informazioni estremamente ridondanti e, quindi, comprimibili. Come sopra esemplificato, la compressione di alcune stringhe rappresentanti alcuni sottoinsiemi di cammini minimi è banalmente attuabile memorizzando solamente, per ogni sottoinsieme, le stringhe di lunghezza maggiore. Nell'esempio, per effettuare una prima compressione, è sufficiente memorizzare solamente le stringhe riportate in grassetto (tabelle 3.1 e 3.3).

Per arrivare a descrivere e ben comprendere la soluzione proposta al capitolo 5, procediamo per gradi, osservando, per prima cosa, le cause della ridondanza, in modo da cercare di risolvere il problema di memorizzazione, attraverso un processo di "divide et impera", che scomponga la struttura grafo e il relativo insieme dei cammini minimi, mantenendo insieme gli elementi che condividono più informazione. In questo modo, possiamo scomporre il problema in sottoproblemi di dimensione minore, risolverli e ricomporre le diverse parti nella soluzione finale (vedi soluzione definita al capitolo 5). Come ben espresso tramite l'esempio, *la ridondanza deriva principalmente dal fatto che tutti i sottocammini di cammini minimi sono anch'essi cammini minimi, ma anche dal fatto che alcuni cammini possono condividere parte del loro percorso.* Nei prossimi paragrafi, formalizzeremo queste due concause in due proprità.

3.1 Sottocammini di cammini minimi

Dopo una prima analisi generale sulla presenza di ridondanza nell'insieme dei cammini minimi, riportiamo alcune proprietà derivate dalle osservazioni effettuate durante la ricerca, accompagnandole con prove ed esempi.

Proprietà 3.1. *Dato un grafo $G = (V, E)$ e un cammino in esso definito P, costituito dalla sequenza di nodi $v_1 \dots v_{k-1}v_k$, P è minimo per la coppia*

di nodi (v_1, v_k)*, se e solo se ogni sottocammino,* $v_i v_{i+1} \dots v_j$*, è anch'esso un cammino minimo in* G *per la coppia* (v_i, v_j)*, per ogni* i *e* j *tali che* $1 \leqslant i < j \leqslant k$*.*

Proprietà che possiamo più semplicemente enunciare nel modo seguente:

Proprietà. *Dato un grafo* $G = (V, E)$ *e un qualsiasi cammino minimo* P*, su esso definito come sequenza nodi, ogni sottosequenza di* P *è anch'essa un cammino minimo.*

Dalla proprietà appena enunciata, possiamo facilmente intuire che, rappresentando un cammino minimo, si rappresentano indirettamente, anche tutti i cammini minimi che sono sottocammini di tale cammino minimo. Per questo motivo la proprietà 3.1 può essere sfruttata per operare una compressione sui cammini minimi: *memorizzando un unico cammino minimo riusciamo implicitamente a rappresentare anche tutti i cammini minimi, sottocammini di esso.* Nell'esempio, possiamo osservare che tutti gli elementi presenti nelle righe con ugual indice delle tabelle (3.1 e 3.2) sono esprimibili tramite il primo elemento della riga (scritto in grassetto) rappresentante il cammino minimo con maggior numero di "hop". In figura 3.2 rappresentiamo, sottoforma di nodi e archi, il cammino minimo corrispondente alla prima riga delle tabelle (3.1 e 3.2) così da visualizzare graficamente la proprietà 3.1.

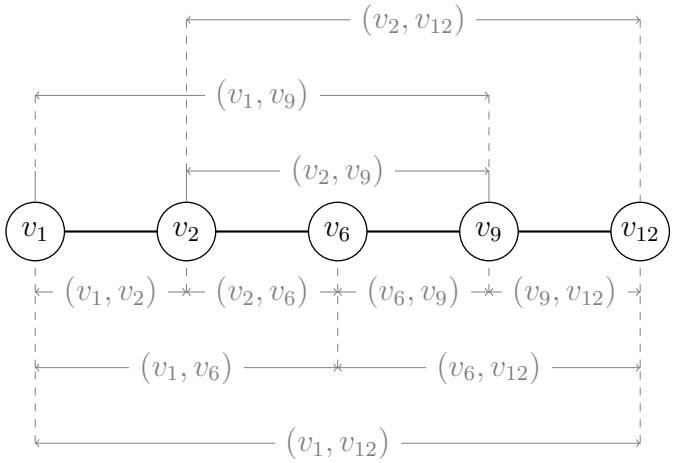


Figura 3.2: Un cammino minimo da v_1 a v_{12} con relativi sottocammini, anch'essi minimi. Memorizzando il cammino minimo comprendente il maggior numero di hop, si riesce a tener traccia anche dei cammini minimi in esso contenuti

Estendendo la proprietà osservata deduciamo che, una prima compressione dell'insieme dei cammini minimi può essere operata memorizzandone un

sottoinsieme proprio. Tale sottoinsieme deve contenere tutti e soli i cammini minimi che non sono sottocammini di altri cammini minimi. Nell'esempio (comprendente le tabelle 3.1, 3.2 e 3.3) questa strategia di compressione si può attuare, rappresentando tutti i cammini minimi attraverso la memorizzazione dei soli cammini minimi evidenziati in grassetto.

3.2 Condivisione di sottocammini

Proprietà 3.2. *Dato un grafo $G = (V, E)$ e due cammini in esso definiti, $\hat{P}$ e $\bar{P}$, costituiti rispettivamente dalle sequenza di nodi $\hat{v_1} \dots \hat{v_{k-1}} \hat{v_k}$ e $\bar{v_1} \dots \bar{v_{j-1}} \bar{v_j}$, $\hat{P}$ e $\bar{P}$ condividono un sottocammino in comune, se e solo se esiste una sequenza X di nodi, $x_1 \dots x_i$ con $i \geq 2$, presente in entrambe le sequenze (di $\hat{P}$ e di $\bar{P}$). Per la proprietà 3.1 la sequenza X forma un cammino minimo da x_1 a x_i.*

Questa proprietà definisce sotto quali ipotesi i cammini minimi condividono parte del proprio percorso. Come la proprietà 3.1, anche questa dipende dalle connessioni del grafo, ma, mentre per definire la prima proprietà non è necessario analizzare le singole connessioni tra nodi (infatti analizziamo i cammini minimi espressi sottoforma di stringhe e relative sottostringhe), per definire, invece, la seconda proprietà dobbiamo scendere al livello dei singoli nodi ed archi per individuare porzioni di cammini comuni a più cammini minimi. Per questo motivo di seguito, andiamo ad analizzare, a livello qualitativo, l'impatto (sull'insieme dei cammini minimi) della *presenza o dell'assenza di un arco* in un grafo e dell'*esistenza di un nodo "interno" comune* a più cammini, individuati tramite una visita BFS[4], a partire da un nodo s.

Presenza (o assenza) di un arco

Come prima osservazione, possiamo notare che, in linea generale, la presenza (o l'assenza) di uno specifico arco aumenta (o riduce) le possibili scelte di "direzione" da un determinato nodo, permettendo (od ostacolando) così la formazione di nuovi cammini. Di conseguenza, ci possiamo chiedere quando il "nuovo" arco entra a far parte di uno o più cammini minimi.

Lemma 1. *Dato un grafo $G = (V, E, W)$ e un grafo $G' = (V, E', W')$, tale che $E' = E \cup \{\bar{e}\}$ ($\bar{e} = (u, v)$, u e $v \in V$) e che W' equivalga alla funzione di costo W estesa al nuovo arco:*

$$\forall e \in E: \quad W'(e) = W'(e) \quad e \quad W'(\bar{e}) = w \tag{3.3}$$

[4]Come osserveremo, visitare i nodi in ampiezza a partire da un nodo s, permette di individuare non soltanto i cammini minimi con sorgente s

allora l'insieme dei cammini minimi di G' è diverso da quello di G se, e solo se, w è minore o uguale al costo del cammino minimo da u a v presente in G.

Dopo aver visto sotto quali ipotesi la presenza, o l'assenza, di un arco modifica l'insieme dei cammini minimi, analizziamo quali elementi dell'insieme dipendono da uno specifico arco. Per far ciò, utilizziamo il concetto di distanza definito al paragrafo 2.1 (definizione 2.0.9).

Lemma 2. *Dato un grafo $G = (V, E, W)$, un grafo $G' = (V, E', W')$, con $E' = E \cup \{\bar{e}\}$ (W' definito come al punto sopra[5]) e scelti arbitrariamente s e $t \in V$, se l'insieme dei cammini minimi da s a t in G è diverso dall'insieme dei cammini minimi tra s e t in G' allora $\bar{e} = (u, v)$ (con u e $v \in V$) è di costo minimo per la coppia (u, v) e, dette $d(s, u)$ e $d(s, v)$ rispettivamente le distanze nel grafo G tra s e u e tra s e v, $|d(s, u) - d(s, v)| \geq W(\bar{e})$.*

Presi s e $t \in V$ si definisca $\mathcal{P}_G^{<}(s, t)$ l'insieme dei cammini minimi, da s a t in G e $\mathcal{P}_{G'}^{<}(s, t)$ l'insieme dei cammini minimi, da s a t in G', riscriviamo il lemma sopra enunciato, come:

$$\mathcal{P}_G^{<}(s,t) \neq \mathcal{P}_{G'}^{<}(s,t) \quad \Longrightarrow \quad \exists\, |d(s,u) - d(s,v)| \geq W(\bar{e}) \quad \wedge \quad d'(u,v) = W(\bar{e})$$

Dimostrazione. $\Rightarrow$
Per la definizione 2.0.6 ogni cammino P in un grafo è determinato da una sequenza di archi, poiché $E' = E \cup \{\bar{e}\}$, segue che:

$\forall P \in \mathcal{P}_G,\ \exists P' \in \mathcal{P}_{G'} : P' = P \quad$ e che $\quad \exists P'' \in \mathcal{P}_{G'} : P'' \notin \mathcal{P}_G$.
$\forall P \in \mathcal{P}_G,\ \exists P' \in \mathcal{P}_{G'} : P' = P \Rightarrow \forall P \in P_G^{<}(s,t),\ \exists P' \in \mathcal{P}_{G'} : P' = P$
$\forall P \in P_G^{<}(s,t),\ \exists P' \in \mathcal{P}_{G'} : P' = P \wedge \mathcal{P}_G^{<}(s,t) \neq \mathcal{P}_{G'}^{<}(s,t)$
$\quad \Rightarrow \exists P \in P_G^{<}(s,t) : P \notin P_{G'}^{<}(s,t) \quad \vee \quad \exists P'' \in P_{G'}^{<}(s,t) : P'' \notin P_G^{<}(s,t)$
Sia $P \in P_G^{<}(s,t) : P \notin P_{G'}^{<}(s,t)$
$\Rightarrow \exists P'' \in P_{G'}^{<}(s,t) : W(P'') < W(P) \wedge \bar{e} \in P''$
Se $\bar{e} \notin P'' \Rightarrow P'' \in P_G(s,t) \Rightarrow P'' \in P_G^{<}(s,t)$, poiché $P'' \notin P_G^{<}(s,t) \Rightarrow \bar{e} \in P''$

Dimostrato che:
$\exists P \in P_G^{<}(s,t) : P \notin P_{G'}^{<}(s,t) \quad \Rightarrow \quad \exists P'' \in P_{G'}^{<}(s,t) : P'' \notin P_G^{<}(s,t)$.
Senza perdere di generalità, consideriamo solamente il caso in cui

$$\exists P'' \in P_{G'}^{<}(s,t) : P'' \notin P_G^{<}(s,t) \wedge \bar{e} \in P''.$$

[5] È importante sottolineare che assumiamo che non esistano archi di costo negativo (o pari a zero)

$\bar{e} \in P'' \wedge \bar{e} = (u, v) \Rightarrow P'' = s, \ldots, u, v, \ldots, t \Rightarrow P''$ è dato dalla concatenazione di un cammino $P_{(s,u)}$ da s ad u, con l'arco $\bar{e}$, a sua volta concatenato con un cammino $P_{(v,t)}$ da v a t. Poiché i cammini minimi sono cammini semplici $\bar{e} \notin P_{(s,u)} \wedge \bar{e} \notin P_{(v,t)} \Rightarrow P_{(s,u)} \in \mathcal{P}_G \wedge P_{(v,t)} \in \mathcal{P}_G$.

$$\begin{aligned} & W(P'') = W(P_{(s,u)}) + W(\bar{e}) + W(P_{(v,t)}) \quad \wedge \\ & \quad W(P'') < W(P), \forall P \in \mathcal{P}_G^{<}(s,t) \quad \wedge \\ & \quad P_{(s,u)} \in \mathcal{P}_G \wedge P_{(v,t)} \in \mathcal{P}_G \\ & \Rightarrow W(\bar{e}) < W(P_{(u,v)}),\ \forall P_{(u,v)} \in \mathcal{P}_G^{<}(u,v) \\ & \Rightarrow W(\bar{e}) = d'(u,v) \end{aligned}$$

Dalla proprietà 3.1 segue che: il cammino dato dalla concatenazione di $P_{(s,u)}$ con l'arco $\bar{e}$ è un cammino minimo, perché sottocammino del cammino minimo P''; da ciò, segue che, in G':
$d'(s,v) = W(P_{(s,u)}) + W(\bar{e}) \neq W(P_{(s,u)}) = d(s,u)$. In G invece $d(u,v)$ era minore o uguale a $W(\bar{e})$, perciò $|d(s,u) - d(s,v)| \geq W(\bar{e})$.

Lemma 3. *Dato un grafo $G = (V, E, W)$, un grafo $G' = (V, E', W')$, con $E' = E \cup \{\bar{e}\}$ (W' definito come al punto sopra) e scelti arbitrariamente s e $t \in V$, se $\bar{e} = (u, v)$ (con u e $v \in V$) è di costo minimo per la coppia (u, v) e $|d(s,u) - d(s,v)| \geq W(\bar{e})$, allora l'insieme dei cammini minimi da s a t in G è diverso dall'insieme dei cammini minimi tra s e t in G'.*

Il lemma 3 corrisponde all'implicazione inversa del lemma 2, ma necessita di due ipotesi aggiuntive:

$$\begin{aligned} \mathcal{P}_G^{<}(s,t) \neq \mathcal{P}_{G'}^{<}(s,t) \quad \Longleftarrow \quad & \exists P' \in P_G^{<}(s,t) : u \in P' \quad \wedge \\ & \exists P'' \in P_G^{<}(s,t) : v \in P'' \quad \wedge \\ & |d(s,u) - d(s,v)| \geq W(\bar{e}) \ \wedge \ d'(u,v) = W(\bar{e}) \end{aligned}$$

Dimostrazione. $\Leftarrow$
Assumere, per ipotesi, che il cammino composto dal singolo arco $\bar{e} = (u, v)$ sia, in G', di costo minimo per la coppia (u, v), equivale a dire che $W(e) =$

$d'(u,v)$, dove $d'(u,v)$ è la distanza tra i due nodi nel grafo G'.

$\exists P' \in P_G^<(s,t) : u \in P' \quad \wedge \quad \exists P'' \in P_G^<(s,t) : v \in P''$
$\wedge \quad$ in G, $|d(s,u) - d(s,v)| \geq W(\bar{e}) \quad \wedge \quad$ in G', $d'(u,v) = W(\bar{e})$
$\Rightarrow$ i cammini minimi P' e P'' possono essere visti rispettivamente come le concatenazioni di due cammini minimi: $P_{(s,u)}$ concatenato a $P_{(u,t)}$ e $P_{(s,v)}$ concatenato a $P_{(v,t)}$.

Senza perdere di generalità, assumiamo che $d_{(s,u)} < d_{(s,v)}$ ciò implica che, poichè $\exists P' \in P_G^<(s,t) : u \in P' \quad \wedge \quad \exists P'' \in P_G^<(s,t) : v \in P''$, $d_{(u,t)} < P_{(v,t)}$ da cui, poiché $\hat{P}$ è dato dalla concatenazione di $P_{(s,u)}$, $\bar{e}$ e $P_{(v,t)}$ segue che : $\exists \hat{P} \in P_{G'}^<(s,t) : \hat{P} \in P_G^<(s,t)$,

□

In figura, riportiamo il caso opposto in cui sia aggiunto in G un arco tra u e v, ma questi si trovino alla stessa distanza da s. Assumendo che non esistano archi di costo negativo (o pari a zero), si può osservare che l'introduzione del nuovo arco (u,v) non ha effetto sull'insieme dei cammini minimi, esistenti tra s e t.

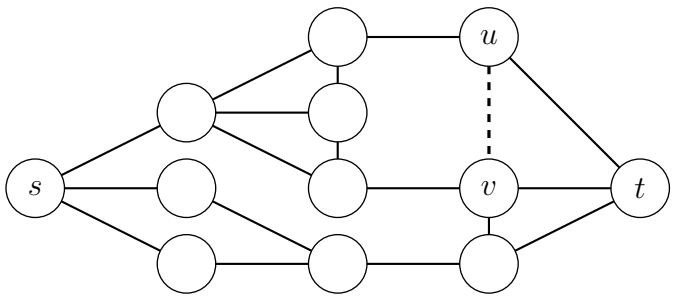


Figura 3.3: Aggiungendo l'arco dal nodo u al nodo v (rispetto al grafo in figura 3.1) i cammini minimi da s a t non subiscono variazioni, poiché u e v si trovano entrambi a distanza 4 da s.

Nodo "interno" comune a più cammini minimi

Un'ulteriore osservazione che possiamo fare, riguarda la condivisione (o la non condivisione) di un nodo da parte di due o più cammini di un grafo non pesato, individuati tramite una visita BFS.

Lemma 4. *Dato un grafo $G = (V, E)$ non pesato e due cammini $\bar{P}$ e $\hat{P}$, in esso definiti rispettivamente dalle sequenze di nodi $\bar{n}_1 \dots \bar{n}_{k-1}\bar{n}_k$ e $\hat{n}_1 \dots \hat{n}_{l-1}\hat{n}_l$ individuate tramite una visita BFS a partire da un nodo x, se esiste un nodo n tale che $n = \bar{n}_i = \hat{n}_j$ per $1 \leq i \leq k$, $1 \leq j \leq l$ ($n \neq t$), allora o i due cammini hanno un sottocammino in comune oppure esistono altri due cammini $\bar{Q}$ e $\hat{Q}$ definiti rispettivamente dalle sequenze di nodi $\bar{n}_1 \dots n\ \hat{n}_h \dots \hat{n}_l$ e $\hat{n}_1 \dots n\ \bar{n}_g \dots \bar{n}_k$, con $h \leq l$ e $g \leq k$.*

Dimostrazione.
Poiché i due cammini $\bar{P} = [\bar{n}_1 \dots\ \bar{n}_{k-1}\bar{n}_k]$ e $\hat{P} = [\hat{n}_1 \dots \hat{n}_{l-1}\hat{n}_l]$ sono individuati attraverso una visita BFS, a partire dal nodo x, e condividono un nodo n, $\bar{P} = \bar{n}_1 \dots \bar{n}_i\ n\ \bar{n}_g \dots \bar{n}_k$ e $\hat{P} = \hat{n}_1 \dots \hat{n}_j\ n\ \hat{n}_h \dots \hat{n}_l$, segue che

$$\begin{aligned}
&\forall c \text{ con } h \leq c \leq l\ ,\ \forall a \text{ con } 1 \leq a \leq i\ ,\ \nexists P' \in P_G^{<}(\bar{n}_a, \hat{n}_c) : \\
&\qquad\qquad W(P') < W(P_{(\bar{n}_a,n)}) + W(P_{(n,\hat{n}_c)}) \text{ e} \\
&\forall c' \text{ con } g \leq c' \leq k\ ,\ \forall b \text{ con } 1 \leq b \leq j\ ,\ \nexists P'' \in P_G^{<}(\hat{n}_b, \bar{n}_{c'}) : \\
&\qquad\qquad W(P'') < W(P_{(\hat{n}_b,n)}) + W(P_{(n,\bar{n}_{c'})})
\end{aligned}$$

Per questo motivo, i cammini dati dalla concatenazione di $P_{(\bar{n}_a,n)}$ e $P_{(n,\hat{n}_c)}$ con $1 \leq a \leq i$ e $h \leq c \leq l$ sono cammini minimi per le coppie di nodi $(\bar{n}_a, \bar{n}_c)$, mentre quelli dati dalla concatenazione di $P_{(\hat{n}_b,n)}$ e $P_{(n,\bar{n}_{c'})}$ con $1 \leq b \leq j$ e $i \leq c' \leq g$ sono cammini minimi per le coppie di nodi $(\hat{n}_b, \bar{n}_{c'})$.

Da notare che nel caso in cui $\bar{P}$ e $\hat{P}$ condividano più nodi tra loro adiacenti, tali cammini hanno un sottocammino in comune. □

Di seguito riportiamo due esempi che illustrano i due diversi casi descritti dal lemma.

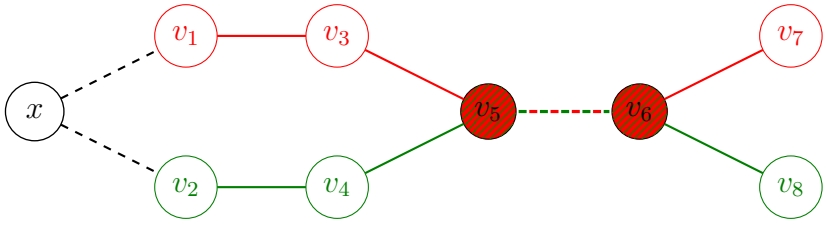
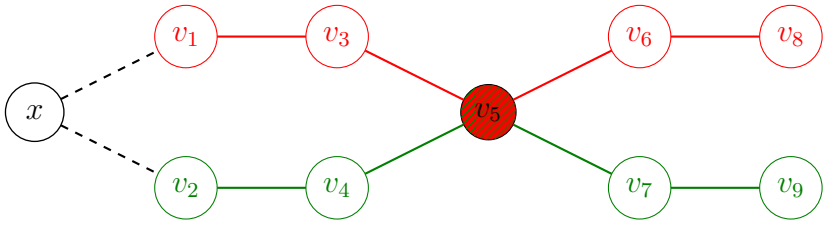
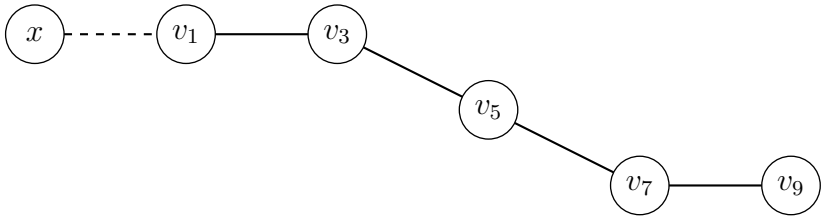


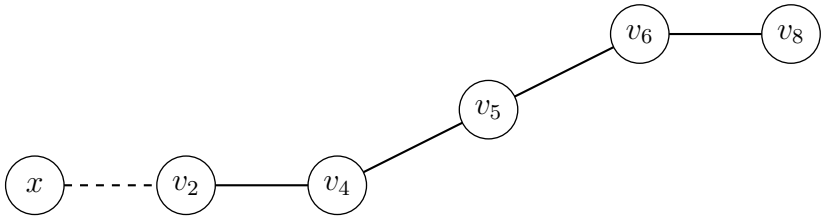
Figura 3.4: Due cammini, generati tramite una visita BFS, a partire dal nodo x, che condividono un sottocammino



(a) Cammini minimi che condividono un nodo



(b) Cammino minimo da v_1 a v_9



(c) Cammino minimo da v_2 a v_8

Figura 3.5: Esistenza di quattro cammini minimi: $\bar{P}$ in verde, $\hat{P}$ in rosso, $\bar{Q}$ trascritto in figura (a) e $\hat{Q}$ in trascritto figura (b). Notare che $\bar{P}$ condivide una porzione di cammino con $\bar{Q}$ e che $\hat{P}$ condivide una porzione di cammino con $\hat{Q}$.

Dalla proprietà osservata si deduce che *in un grafo non pesato*, un'ulteriore compressione dell'insieme dei cammini minimi può essere operata utilizzando una struttura che rappresenti i diversi cammini minimi, generati tramite una visita BFS, condividendo le porzioni di cammini comuni tra di essi. Nell'esempio (comprendente le tabelle 3.1, 3.2 e 3.3) questa strategia di compressione porterebbe a far condividere l'ultima parte della sequenza, nello specifico i nodi $v_6v_9v_{12}$, ai due cammini $v_1v_2v_6v_9v_{12}$ e $v_5v_6v_9v_{12}$.

Capitolo 4

Rappresentare cammini minimi

Nel seguente capitolo, inquadriamo meglio il problema che ci siamo proposti di risolvere, descrivendo possibili soluzioni e analizzandone il costo in termini di tempo e spazio. Iniziamo a descrivere le soluzioni più semplici e non efficienti, per poi migliorarne le prestazioni. Da notare che, poiché ci proponiamo di rappresentare e recuperare tutti i cammini minimi per una data coppia di vertici, ci possiamo aspettare di pagare maggiormente (in termini di tempo) se il numero dei cammini da recuperare è maggiore. Ci proponiamo di sviluppare un algoritmo il cui costo dipende dal numero di valori restituiti (output sensitive): ci è quindi "permesso" di impiegare un numero di passi proporzionale al numero di cammini minimi esistenti e alla loro lunghezza. Non riterremo soddisfacente un algoritmo che paghi $\mathcal{O}(n)$ passi per recuperare un singolo cammino minimo di lunghezza pari a l, con $l \ll n$.

Riportiamo quindi quattro possibili soluzioni per il problema da noi affrontato, comparandole tra loro e soffermandoci sui diversi costi e sull'evidente tradeoff che si viene a creare tra la memoria occupata e il tempo necessario alla "decompressione" dell'informazione richiesta, cioè al recupero dei cammini minimi. Possiamo ben osservare che, a un minor tempo di elaborazione dell'algoritmo, corrisponde una maggior occupazione in memoria e viceversa. Questo andamento inverso delle due grandezze (spazio occupato e tempo impiegato) è conseguenza diretta del fatto che una maggior compressione delle informazioni necessita maggior tempo in decompressione e viceversa. Questo fenomeno si verifica in tutti i casi in cui ci si prefigge di comprimere un contenuto informativo e successivamente di recuperarlo (si veda per esempio [26] illustrante il gap tra spazio e tempo per il problema SSSP). È quindi essenziale trovare il giusto gap tra *spazio occupato* e *tempo di elaborazione*.

4.1 Soluzioni limite

Riportiamo le due soluzioni limite, che si posizionano ai due estremi, come occupazione di spazio e impiego di tempo. Osservando tali soluzioni, possiamo constatare che, cercando di linearizzare lo spazio occupato, aumenta esponenzialmente il tempo necessario a decomprimere l'informazione; mentre, rendendo costante il tempo di recupero dell'informazione, "esplode" la quantità di memoria utilizzata. Nonostante queste due soluzioni non siano accettabili, per la loro eccessiva complessità, è tuttavia importante considerarle, per meglio comprendere la dimensione del problema e l'intervallo in cui si può andare a collocare una soluzione ragionevole in termini di costi.

Gli algoritmi sottostanti si occupano di recuperare tutti i cammini minimi in un grafo non pesato e non diretto, data una qualsiasi coppia di nodi.

4.1.1 Spazio lineare

Una prima possibile soluzione consiste nel recuperare tutti i cammini minimi (per una specifica coppia di vertici) non operando nessun preprocessing sul grafo dato. La memoria occupata consiste della sola struttura di grafo ed ha quindi costo lineare[1], mentre il costo di recupero dell'informazione è esponenziale. Non avendo strutture dati aggiuntive al grafo o altre informazioni di supporto alla ricerca dei cammini minimi, la soluzione più ragionevole, per recuperare i cammini di lunghezza minima per una coppia di vertici (u, v), è muoversi in ampiezza lungo il grafo iniziando a partire da uno dei due nodi e ricercando l'altro.

Ricordiamo che il grafo su cui effettuiamo la ricerca è un grafo non pesato e non diretto, e che stiamo recuperando i soli cammini minimi da u a v; per questo motivo, il minimo numero di hop, che intercorre tra i due vertici, equivale alla loro distanza e la strategia di ricerca migliore consiste nella BFS (v. def. 2.0.11 al cap. 2). Così facendo, nel caso esistano cammini non minimi tra u e v non li visiteremo.

Questa soluzione ha lo svantaggio di effettuare la ricerca senza avere una "bussola", che indichi se il cammino, che si sta percorrendo, sia un potenziale cammino minimo per (u, v). L'unica informazione che può arrestare la ricerca, senza proseguire ad analizzare tutti i cammini del grafo, è data dalla distanza tra il nodo da ricercare e il nodo di partenza, informazione che acquisiamo nel momento in cui individuiamo il primo cammino minimo tra u e v. Se la distanza tra i vertici risulta essere uguale a d sappiamo che non

[1]Si ricorda che la complessità lineare viene riferita al costo $\mathcal{O}(n + m)$, che considera la somma del numero dei nodi n e degli archi m

dobbiamo visitare tutti i nodi che hanno una distanza da u (nodo di partenza della visita) maggiore di d. Di seguito, riportiamo un esempio che evidenzia il costo esponenziale di ricerca, al caso pessimo.

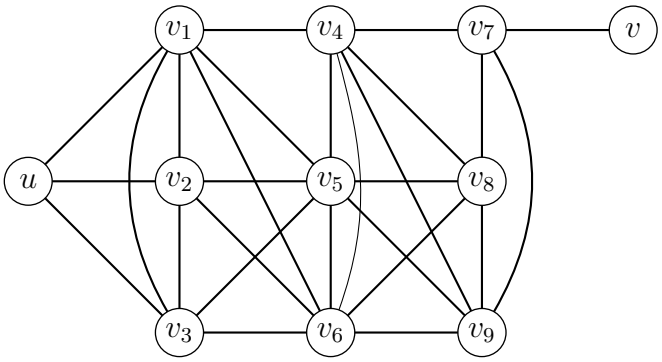


Figura 4.1: Caso pessimo per la soluzione che usa spazio lineare

Nel grafo in figura 4.1 esiste un unico cammino minimo dal vertice u al vertice v; ma se attuiamo l'algoritmo di risoluzione sopra proposto per individuare tutti i cammini minimi per (u, v), dobbiamo effettuare $\mathcal{O}(\sqrt{n}^{\sqrt{n}})$ passi. Infatti, trascurando il nodo sorgente e il nodo target, i nodi sono distribuiti in numero equo su $\sqrt{n}$ livelli, ciascuno dei quali è formato dai nodi che hanno la stessa distanza da u. Al contempo, soltanto il primo nodo di ogni livello è connesso, sia a tutti i nodi appartenenti al suo stesso livello, che a quelli del livello successivo; gli altri hanno gli stessi archi, se si esclude quello che li collega al primo vertice del livello successivo. Possiamo valutare il numero di passi effettuati dall'algoritmo di ricerca applicato al grafo descritto, considerando ogni cammino visitato con la ricerca BFS, come una possibile disposizione con ripetizioni di lunghezza l (distanza tra u e v) di un insieme di $\sqrt{n}$ oggetti (nodi). Da ciò otteniamo il costo pari a $\mathcal{O}(\sqrt{n}^{\,l})$ che, al caso pessimo, equivale a $\mathcal{O}(\sqrt{n}^{\sqrt{n}})$. Oer questo motivo, algoritmo non solo non risulta output sensitive ma, al caso pessimo, per recuperare un unico cammino minimo, paga un costo esponenziale in n.

4.1.2 Tempo costante

L'algoritmo di risoluzione proposto al passo precedente compie "l'errore" di non tenere informazioni aggiuntive al grafo. In tal modo comprime al massimo i cammini minimi del grafo, usando il grafo stesso[2]. Mantenendo in

[2]notare che il grafo contiene tutte le informazioni necessarie ad individuare i cammini minimi e non contiene dati aggiuntivi, dal momento che il grafo non è pesato e, di conse-

mememoria solo il grafo, ci ritroviamo a dover calcolare ogni volta i cammini minimi, attuando una visita BFS non guidata del grafo.

Come soluzione opposta alla precedente, possiamo decidere di tenere l'informazione in modo esplicito, mantenendo in memoria tutti i cammini minimi esistenti tra tutte le coppie di vertici del grafo. In tal modo, costruendo su di esse un dizionario, che abbia per chiave una coppia di nodi, possiamo recuperare i cammini minimi richiesti, in tempo di acceso costante $\mathcal{O}(1)$, poiché questi sono memorizzati in corrispondenza di un'unica chiave. In questo caso è lo spazio ad "esplodere" esponenzialmente rispetto al numero di nodi, poiché non ci preoccupiamo di memorizzare informazioni tra loro molto "ridondanti".

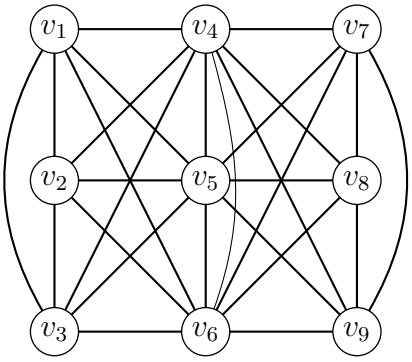


Figura 4.2: Caso pessimo per la soluzione che impiega tempo costante

Considerando il grafo in figura 4.2 e applicandovi il metodo di risoluzione appena descritto, otteniamo un dizionario grande $\sum_{i=2}^{\sqrt{n}} \sum_{j=0}^{\sqrt{n}-i+1} \sqrt{n}^{\,i}$; questo accade poiché detto metodo mantine tutti i cammini minimi, i quali possono avere lunghezza che varia da uno a $\sqrt{n}$, a seconda delle posizioni relative del nodo sorgente (corrispondente all'indice j) e del nodo destinazione (indice i).

Riportando la sommatoria più interna a livello di prodotto, otteniamo:

$$\sum_{i=2}^{\sqrt{n}} \sum_{j=0}^{\sqrt{n}-i+1} \sqrt{n}^{\,i} = \sum_{i=2}^{\sqrt{n}} (\sqrt{n} - i + 1) * (\sqrt{n}^{\,i}) > \sqrt{n}^{\sqrt{n}}$$

il costo in spazio è quindi esponenziale rispetto al numero di nodi.

guenza, ogni arco sarà contenuto in almeno un cammino minimo (il cammino che unisce i due vertici agli estremi dell'arco)

4.2 Verso la soluzione

Dopo aver analizzato le soluzioni limite per il problema del recupero di tutti i cammini minimi per una data coppia di vertici, riportiamo due soluzioni che ci avvicinano alla soluzione da noi elaborata.

La prima soluzione introduce l'idea di ricercare i cammini minimi solamente in una specifica porzione di grafo dipendente dal nodo da cui iniziamo la ricerca (tale porzione di grafo consisterà in un DAG). Questa riduzione dello spazio di ricerca dipende dal fatto che, dato un grafo $G = (V, E)$, fissato un nodo $u \in V$ e detto E' l'insieme degli archi utilizzati nei cammini minimi da u a v, $\forall v \in V$, $E' \subseteq E$.

La seconda soluzione mostra l'utilità di avere una "bussola" che ci permetta, ad ogni passo di ricerca, di determinare se la direzione di ricerca scelta sia giusta o sbagliata. Inoltre, osserveremo che, se le direzioni possibili sono molte, sapere se la direzione scelta è giusta o meno, dopo averla percorsa, non è sufficiente a raggiungere l'obiettivo che ci siamo posti: essere output sensitive. Per questo motivo, introdurremo una nuova tipologia di "bussola" che ci "indichi" la direzione da percorrere, così da non dover "imboccare" cammini per poi accorgersi che sono errati, cioè che non portano al target o che non sono minimi. Questa seconda "bussola" è rappresentata dal Bloom filter (v. def. al cap. 2), di cui parleremo più ampiamente nel paragrafo ad esso dedicato (4.2.2).

4.2.1 Scomporre il problema

Una possibile soluzione al problema di recuperare tutti i cammini minimi per una data coppia di nodi, è data dall'utilizzo di n DAG, ognuno dei quali "radicato" in un diverso vertice del grafo. Tali DAG sono costruiti attraverso una visita BFS, mantenendo una lista di predecessori per ogni nodo incontrato[3]. In questo caso, l'individuazione dei cammini minimi, per una data coppia di nodi (u, v), si ottiene effettuando una visita BFS del DAG "radicato" in u (o in v visto, che il grafo non è orientato).

Si può osservare che il caso pessimo, per questo algoritmo (figura 4.3), è rappresentato da una biclique (un grafo "completo" bipartito). Il costo della visita necessaria a recuperare i cammini minimi non risulta eccessivo: al caso pessimo, equivale al numero di vertici, più il numero di archi del DAG, $\mathcal{O}(m + n)$. Ad essere troppo grande, in questo caso, è lo spazio occupato per

[3]Il metodo è di norma descritto per grafi pesati e, di conseguenza, utilizza per la visita l'algoritmo Dijkstra

la memorizzazione degli n DAG, ognuno dei quali occupa $\frac{n}{2} + \frac{n}{2} \cdot \frac{n}{2} = \mathcal{O}(n^2)$. In totale gli n DAG occupano $\mathcal{O}(n^3)$.

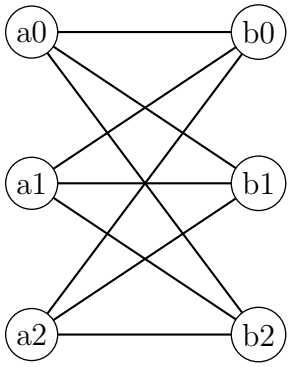


Figura 4.3: Biclique: caso pessimo per la soluzione che usa n DAG

È altresì importante segnalare che, nel caso di grafi sparsi, l'uso di DAG costruiti tramite n visite (una per ogni nodo) rappresenta una buona strategia risolutiva per il problema considerato.

4.2.2 Usare una "bussola"

Considerando quanto sopra, la soluzione, con il miglior tradeoff tra spazio occupato e tempo impiegato a decomprimere le informazioni, si può trovare riducendo lo spazio occupato dalle strutture aggiuntive create in fase di preprocessing, cercando, al contempo, di non aumentare eccessivamente il tempo di elaborazione. Nel caso precedente, le strutture aggiuntive, che ci guidano nella ricerca, sono costituite dai DAG. Possiamo però pensare di fare un preprocessing che, dato un grafo G, crei strutture di dimensione minore rispetto ai DAG, ma che svolgano lo stesso compito. Scegliamo quindi di costruire, durante il preprocessing, solamente una "bussola", che ci guidi attraverso G nella ricerca dei cammini minimi.

"Bussola ritardata"

La prima opzione consiste nell'utilizzo, come "bussola", della matrice delle distanze, determinata utilizzando n volte l'algoritmo di Bellman Ford o l'algoritmo di Floyd-Warshall, a seconda della densità del grafo G (v. par. 2.3). La matrice delle distanze occuperà $\mathcal{O}(n^2)$.

Detta *dist* la matrice delle distanze, t il target da raggiungere tramite cammini minimi, y il nodo in cui ci triamo ad un certo passo della visita BFS e x un nodo da cui proveniamo (predecessore di y), sarà possibile utilizzare *dist* come bussola nel seguente modo:

Controllo durante la navigazione: verifica della correttezza del cammino intrapreso

```
                ▷ Se il vertice in cui ci troviamo è più vicino di uno al target
                                    ▷ rispetto al vertice predecessore di esso
```
if $dist[x][t] - dist[y][t] == 1$ **then**
    Allora proseguiamo il cammino
else
    Abbandoniamo il cammino che stiamo visitando
end if

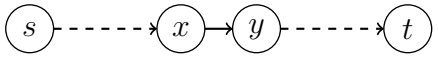


Figura 4.4: Visualizzazione della posizione relativa dei nodi s, x, y e t, dove s è il nodo sorgente da cui inizia la visita BFS

Dobbiamo notare che la "bussola funziona" in modo ritardato, segnalandoci se il passo appena compiuto fa parte di un cammino minimo ricercato o se dobbiamo abbandonare il cammino che stiamo percorrendo, poiché non minimo. Questo ritardo fa sì che si inizi a considerare cammini che successivamente si rivelano non minimi. Queste operazioni inutili hanno un costo che si rivela particolarmente oneroso, nel caso in cui il numero di cammini minimi sia molto minore rispetto al numero di cammini iniziati a visitare. Le prestazioni, in termini di tempo, peggiorano ulteriormente rispetto all'output, se consideriamo i casi in cui i cammini minimi, non solo siano pochi in numero, ma siano anche brevi, cioè siano composti da un numero di archi m', tale che $m' \ll |V| = n$. Per visualizzare il caso sopra esposto, consideriamo il grafo in figura 4.5.

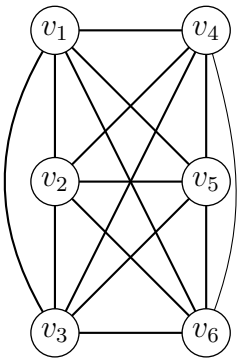


Figura 4.5: Caso pessimo per la soluzione che usa la "bussola ritardata"

In questo caso, tutti i nodi sono tra loro connessi. Se ricerchiamo tut-

ti i cammini minimi tra un qualsiasi nodo della "colonna a sinistra" e un qualsiasi nodo della "colonna a destra", avere la "bussola ritardata" non ci aiuta. Infatti, durante la ricerca BFS, la "bussola ritardata" dà informazioni a partire dal secondo livello, ma il passo di ricerca iniziale continua ad essere fatto "alla cieca", rischiando così di visitare tutti i nodi, come accade nel caso esposto in figura 4.5. Nel caso specifico, al caso pessimo, visitiamo n nodi per recuperare un unico cammino minimo di lunghezza pari a uno.

"Bussola probabilistica"

Per sopperire alle mancanze della "bussola ritardata" utilizziamo un diverso strumento di instradamento: il Bloom filter (v. sez. 4.2). Poiché tale struttura di dati viene usata per testare se un dato elemento è, o meno, presente in un insieme, possiamo utilizzarla per testare rapidamente, se un certo arco porta verso un dato nodo, attraverso un cammino minimo.

Considerando ogni vertice del grafo, comprensivo della sua stella uscente, costruiamo un filtro di Bloom gerarchico che funga da bussola per lo specifico vertice. Ogni Bloom filter gerarchico è formato da più Bloom filter disposti ad albero binario. Tale disposizione permette di effettuare una ricerca binaria sull'albero. Per far ciò, costruiamo l'albero a partire dalle foglie e, man mano che saliamo di livello nell'albero, fondiamo coppie di nodi del livello precedente. Ogni foglia dell'albero corrisponde ad un arco del vertice (per cui si sta costruendo il filtro gerarchico) ed è costituita da un Bloom filter, rappresentante l'insieme dei nodi, raggiungibili per quello specifico arco. Per individuare quali nodi inserire nel Bloom filter basta utilizzare la matrice delle distanze, controllando se, percorrendo uno specifico arco, ci avviciniamo o meno ad un dato nodo (vedi controllo "Bussola ritardata"). Di conseguenza, per costruzione, ogni nodo interno dell'albero corrisponde ad un sottoinsieme di archi ed è formato da un Bloom filter, rappresentante l'insieme dei nodi raggiungibili, percorrendo anche un solo arco di tale sottoinsieme. Costruiti i bloom filter non abbiamo più bisogno di mantenerci la matrice delle distanze, poiché gli stessi Bloom filter svolgeranno il compito di bussola, con il vantaggio di indicarci se un dato arco faccia parte di un cammino minimo (ricercato), senza doverlo percorrere.

È però importante osservare che, essendo il Bloom filter una struttura di dati probabilistica, è soggetto ad errori (vedi sottoparagrafo 2.4.3), che tuttavia possono essere ridotti, aumentando la dimensione del filtro. Supponendo di scegliere la dimensione dei filtri di Bloom in modo da controllare i falsi positivi, la seguente soluzione risulta essere la migliore tra quelle proposte in questo capitolo. Infatti, l'uso dei Bloom filter (opportunamente costruiti attraverso la matrice delle distanze) permette di recuperare i cammini minimi

per una data coppia di nodi (s, t), in tempo proporzionale al loro numero e alla loro lunghezza.

Per quanto rigurda lo spazio occupato, in questo caso, è necessario mantenere in memoria solamente il grafo G (oggetto della ricerca) e i filtri di Bloom gerarchici, uno per ogni vertice $v \in G$. Ogni filtro di Bloom occupa un numero di bit proporzionale ad n (numero di nodi del grafo) precisamente $Mbit$, con $M = -\frac{n \ln p}{(\ln 2)^2}$, e poiché esiste un filtro per ogni vertice, la memoria occupata è pari a $M = -\frac{n^2 \ln p}{(\ln 2)^2}$ *bit*. In totale, questa soluzione occupa $\mathcal{O}(m+n)$, per la memorizzazione del grafo e $\mathcal{O}(n^2 \ln p)$ *bit* per quella dei filtri di Bloom.

Capitolo 5

Un nuovo metodo di rappresentazione

Dopo aver esaminato il problema nel suo complesso ed aver osservato che la qualità di un algoritmo (nello specifico, di un algoritmo di shortest-path query) è data dal tradeoff, esistente tra lo spazio utilizzato per rappresentare l'informazione e il tempo necessario a recuperarla, nel seguente capitolo proponiamo una soluzione alternativa, che migliori le prestazioni di quelle precedentemente esaminate (capito 4). Per fare questo sviluppiamo un metodo alternativo di rappresentazione implicita di tutti i cammini minimi di un grafo, non diretto e non pesato, per una data coppia di vertici. Per cercare di ottenere un buon algoritmo, in termini di costo in spazio e tempo, utilizziamo una struttura di dati che sfrutti le proprietà di ridondanza osservate nel capitolo 3 e che, allo stesso tempo, tenga conto delle osservazioni fatte sulle diverse soluzioni proposte nel capitolo 4.

L'ultima soluzione, proposta nel capitolo 4, riconduce il problema di shortest-path query ad un problema di routing. In tal modo, come abbiamo visto, individuare un cammino minimo corrisponde a percorrere un cammino, passando da un nodo al successivo, guidati da una bussola probabilistica costruita in fase di preprocessing. Questa soluzione mantiene in memoria il solo grafo iniziale e un Bloom Filter gerarchico per ogni nodo (per l'uso del filtro di Bloom per il problema di routing si veda [16]).

Tuttavia, anche se detta soluzione risolve il problema di individuare tutti i cammini minimi a costi ragionevoli di tempo e spazio, ci possiamo chiedere se esistano soluzioni alternative, che sfruttino le connessioni del grafo e che non si limitino a creare una "bussola probabilistica" locale. Faremo in modo che il Bloom Filter, di uno specifico nodo, non abbia la necessità di conoscere le direzioni da seguire per un qualsiasi nodo del grafo, ma che conosca tutte e sole le direzioni verso alcuni "nodi ponte". La corrispondenza tra ogni

coppia di vertici sorgente-destinazione e i "nodi ponte" sarà mantenuta in una tabella di routing.

5.1 Soluzione

La soluzione alternativa individuata non può quindi fare a meno del Bloom Filter per l'instradamento, anche se, non si esclude, che, eventuali studi futuri possano individuare strategie di scomposizione del problema, tali da ripartire il numero di connessioni in modo equo tra le diverse porzioni del grafo, cercando così di evitare, o ridurrre, l'utilizzo del Bloom Filter. Ricordiamo che il filtro di Bloom è stato introdotto per diminuire l'eccessivo costo computazionale di ricerca causato dalla presenza di porzioni di grafo fortemente connesse (sottoparagrafo 4.2.2).

Alla base dell'attuale soluzione, principale oggetto di questa tesi, c'è l'idea di determinare i cammini minimi per una certa coppia di vertici (u, v) di un grafo G dato, combinando i risultati di sottoproblemi di dimensione minore, ottenuti scomponendo ricorsivamente il grafo G, tramite un processo di **"divide et impera"**. La combinazione dei risultati intermedi è attuata tramite degli specifici nodi, che chiameremo "nodi ponte", per il ruolo di collegamento che svolgeranno.

5.1.1 Strategia di suddivisione

Il primo problema, che ci troviamo ad affrontare, è quello di definire una strategia di scomposizione del grafo. Le strategie possibili possono essere molteplici: scegliere le componenti maggiormente connesse; fare una scelta guidati dal grado dei nodi; etc.

La strategia da noi scelta è, in parte, guidata dalle proprietà osservate al capitolo 3.

Considerando le proprietà di "ridondanza" descritte nel capitolo 3, optiamo per la divisione del grafo in DAG ottenuti tramite visite BFS.

Come era stato fatto nella soluzione che costruiva un DAG per ogni nodo (sottoparagrafo 4.2.1), cerchiamo di ricondurre la ricerca dei cammini minimi alla sola visita di una (o più) porzioni del grafo.

Scelta la struttura di dati alla base della nostra scomposizione (il DAG) e la strategia di costruzione dello stesso (una visita BFS a partire da un dato nodo), per ogni passo ricorsivo è necessario determinare quale sia il nodo dal quale iniziare a costruire il DAG e su quale sottoinsieme di nodi costruirlo.

Tale nodo è scelto tra tutti i nodi *Peers* appartenenti allo stesso livello[1] (si veda l'algoritmo 4 nel paragrafo 5.3).

Per la scelta del nodo ci "vengono in aiuto" le proprietà dei cammini minimi individuate al capitolo 3. Infatti, cercheremo di sfruttare la forte ridondanza tra gli elementi dell'insieme dei cammini minimi, cercando di memorizzare, in ogni singolo DAG, il maggior numero di cammini minimi. Per questo motivo, poiché dalla proprietà 3.1 deriva che ogni sottocammino di un cammino minimo è anch'esso un cammino minimo, iniziamo la costruzione di ogni DAG a partire da un nodo che abbia diametro massimo rispetto ai nodi dello stesso livello, su cui operiamo il passo ricorsivo (definito dall'algoritmo 6 nel paragrafo 5.3). Per la definizione di diametro si veda la definizione 2.0.10.

Iniziare la visita BFS, a partire dal nodo x con diametro massimo, offre il beneficio di poter implicitamente *rappresentare un maggior numero di cammini minimi, come sottocammini del "più lungo" cammino minimo* esistente per x.

La scelta di creare ogni DAG a partire da x apporta anche altri vantaggi. Uno di questi vantaggi, del quale però non riusciamo a dare un valore quantitativo, è dovuto al lemma 4, descritto nel capitolo 3, che rivela una diretta proporzionalità tra il numero di connessioni e il numero di cammini minimi, generati attraverso una visita BFS. Tale vantaggio consiste nel fatto che rappresentando i cammini minimi di lunghezza maggiore, non solo riusciamo a rappresentare tutti i loro sottocammini, ma *aumentiamo la probabilità di rappresentare porzioni di altri cammini minimi.* Così facendo, possiamo sfruttare la struttura, che stiamo costruendo, per rappresentare alcuni cammini minimi tramite la composizione di sottocammini di altri cammini minimi, attuata attraverso l'uso dei nodi ponte b della tabella di routing *index* (vedi algoritmo 12 nel capitolo 5.3).

Un altro vantaggio, dato dalla decisione di costruire i DAG a partire dai nodi di diametro massimo, deriva dal fatto che, i sottoinsiemi di nodi su cui effettuiamo i passi ricorsivi sono formati raggruppando i nodi, in base alla distanza che hanno rispetto ad s, nodo di partenza del DAG costruito precedentemente (si veda la dichiarazione dell'algoritmo 5 nel capitolo 5.3). Di conseguenza, il numero di porzioni in cui sarà suddiviso ogni DAG (e sulle quali saranno effettuati i passi ricorsivi), sarà minore o uguale al valore del diametro, meno uno. Ogni porzione, in cui sarà suddiviso il DAG, sarà costituita dai nodi equidistanti da s, denominati nodi *Peers* (si veda l'algoritmo 5 e l'algoritmo 6 nel paragrafo 5.3). Così operando, cerchiamo di massimiz-

[1] Ogni livello del DAG è formato dai nodi equidistanti dal nodo da cui inizia la visita BFS.

zare il numero di porzioni in cui scomporre il grafo e, allo stesso tempo, di raggruppare tra loro i nodi che formano un maggior numero di cammini.

Poiché il nostro scopo è quello di ridurre lo spazio utilizzato, operando le scelte sopra descritte, cerchiamo di ridurre il numero di ricorsioni annidate, che equivale al numero di copie che dobbiamo mantenere per uno stesso nodo del grafo (nella soluzione descritta nel sottoparagrafo 4.2.1 il numero di copie per ogni nodo è pari a n, poiché, in quel caso, si utilizzano n DAG, ognuno dei quali contenente tutti i vertici n del grafo).

Andiamo, quindi, a meglio specificare la struttura di rappresentazione dei cammini minimi, la modalità di costruzione della stessa e i passi necessari al recupero dei cammini minimi per una data coppia di vertici, specificati in un'interrogazione.

5.1.2 Strutture di compressione

La struttura di dati, alla base della memorizzazione dei cammini minimi, è il **DAG** (vedi definizione 4.1.1). La compressione dell'informazione deriva dall'utilizzo di un numero finito di DAG per rappresentare l'insieme $DAGs$ di tutti i cammini minimi per tutte le coppie di nodi presenti in un grafo (vedi riga 4 dell'algoritmo 2 nel capito 5.3).

Attuando più visite BFS tra loro annidate, suddividiamo il grafo in DAG, permettendo che ogni nodo possa essere presente in più DAG tra loro annidati, ma facendo corrispondere, ad ogni arco, uno ed un solo DAG.

Mentre si attua la suddivisione in DAG, ci si preoccupa anche di creare una **tabella di routing** (denominata *index* all'interno degli algoritmi, nel paragrafo 5.3). Tale tabella è realizzata tramite un dizionario, che mantiene le informazioni necessarie alla ricostruzione dei cammini minimi, cioè alla "decompressione" delle strutture e alla concatenazione di cammini minimi. Per far ciò, *index* memorizza opportuni nodi (detti "nodi ponte") per ogni possibile coppia di vertici, sorgente-destinazione.

Il dizionario tiene traccia di quali DAG contengono i cammini minimi (o porzioni di essi) esistenti tra una specifica coppia di nodi, usata come chiave del dizionario. Durante la costruzione di ogni DAG (e tramite una successiva visita dello stesso in verso opposto) per ogni nodo x del DAG costruiamo un **Bloom filter** BF (vedi algoritmo 11), che mantiene la corrispondenza tra ogni arco del DAG, radicato in x, ed i "nodi ponte" raggiungibili tramite esso.

Alla fine della fase di preprocessing (cioè di costruzione delle strutture necessarie al mantenimento e al recupero dei cammini minimi) ogni DAG rappresenta uno specifico sottoinsieme dell'insieme dei cammini minimi, mentre,

ogni cammino minimo è ottenuto componendo uno o più cammini minimi, rappresentati rispettivamente in uno o più DAG.

5.1.3 Creazione delle strutture di compressione

DAG annidati

Essendo ogni DAG il risultato di una visita BFS a partire da uno specifico nodo s (si veda algoritmo 5 e 6 nel paragrafo 5.3), chiameremo tale nodo "radice" del DAG. L'algoritmo di preprocessing fa sì che ogni visita BFS crei un DAG (non un albero, come comunemente si procede nelle visite BFS). Ogni DAG è ottenuto tramite la fusione dei nodi dell'albero BFS, che corrispondono ad uno stesso nodo all'interno del grafo (si vedano algoritmi 5 e 6 rispettivamente alle righe 19-29 e 21-31).

Le diverse visite (e i relativi DAG) sono "troncate" in base alle informazioni recuperate nelle visite precedenti, così da non inserire, in più DAG, la stessa informazione e non aggiungere dati "ridondanti". In questo modo, ogni visita BFS è effettuata su una specifica porzione del grafo e l'insieme degli archi è suddiviso tra tutti i DAG.

La struttura di compressione è, quindi, costruita, effettuando più visite BFS del grafo, a partire da un numero finito di nodi "radice" s (visite descritte negli algoritmi 5 e 6 nel paragrafo 5.3). La costruzione termina quando tutti i cammini minimi sono memorizzati. Ogni cammino minimo può essere memorizzato o in modo esplicito, o come sottocammino di un cammino minimo, oppure attraverso la composizione di più porzioni di cammini minimi rappresentati esplicitamente. Per la "ricomposizione" dei cammini minimi, viene utilizzata la tabella di routing *index* (si veda algoritmo 12 nel paragrafo 5.3). Al termine delle visite (dopo aver "coperto" tutti i cammini minimi) la struttura di compressione creata è costituita da più DAG tra loro annidati.

I DAG creati sono tra loro annidati, perché sono il risultato di più chiamate ricorsive, effettuate su porzioni di grafo sempre minori (si veda la riga 14 dell'algoritmo 5 e la riga 15 dell'algoritmo 6, nel paragrafo 5.3). Tali porzioni sono definite partizionando la porzione del livello precedente, corrispondente ad uno specifico DAG. Ogni partizione di un DAG è formata da tutti e soli i nodi, che si trovano ad una stessa distanza dal nodo "radice" del DAG.

Per meglio comprendere la scomposizione dell'insieme dei vertici del grafo in sottoinsiemi, denominati *Peers*, oggetti delle chiamate ricorsive, possiamo considerare l'albero di ricorsione. Ogni nodo di tale albero identifica il sottoinsieme su cui si effettua la ricorsione, sottoinsieme che viene ricorsivamente suddiviso nei figli. Un nodo di un albero non è più soggetto al processo ricorsivo e diventa quindi una foglia dell'albero, se tutti i vertici contenuti nel

sottoinsieme da esso rappresentato corrispondono, nel grafo originale, a nodi tra loro disconnessi. Un esempio di tale albero è riportato nel sottoparagrafo 5.2.2 in fugura 5.13.

Fino a questo momento, per poterci concentrare sull'annidamento dei DAG, abbiamo assunto che ogni visita BFS crei un unico DAG. In realtà, vengono creati due DAG speculari, costituiti da stessi nodi e stessi archi, ma, questi ultimi, con verso opposto. I due DAG ($DAG1$ e $DAG2$) vengono creati contemporaneamente, durante la visita (righe 19-29 e 21-31 rispettivamente negli algoritmi 5 e 6).

Tabella di routing e Bloom Filter

La struttura dei DAG annidati memorizza ogni vertice, tante volte, quante sono le ricorsioni effettuate sui sottoinsiemi, che contengono il vertice stesso. Di contro, questa struttura fa sì che ogni arco del grafo sia mantenuto in memoria, una ed una sola volta. Poiché il grafo considerato non è pesato, ogni arco fa parte di almeno un cammino minimo (quello tra i due estremi dell'arco stesso). Per questo motivo, è necessario che, nei DAG, siano memorizzati tutti gli archi.

Data una coppia di nodi (x, y), il recupero dei cammini minimi, esistenti per tale coppia, consiste in una visita guidata attraverso i diversi DAG costruiti (recupero effettuato tramite l'algoritmo 12 nel paragrafo 5.3). Affinché sia possibile visitare solamente le porzioni dei DAG utili alla costruzione dei cammini ricercati, ad ogni passo è necessario conoscere: il DAG che stiamo visitando (univocamente determinato dal nodo s dell'algoritmo 12), la prossima destinazione y e il/i nodo/i ponte b da raggiungere (cioè le direzioni in cui procedere per raggiungere il nodo considerato y). Mentre la destinazione finale ci è data al momento dell'interrogazione (cioè della richiesta di recupero dei cammini minimi, dal nodo sorgente s, al nodo destinazione t), le informazioni riguardanti la visita ci sono fornite dalla tabella di routing e dai filtri di Bloom.

La tabella di routing ci indica da quale nodo, (o da quali nodi, ognuno dei quali, appartenente ad uno specifico DAG) iniziare, o continuare, la ricerca e quale sia/siano il/i successivo/i nodo/i ponte da raggiungere. I filtri di Bloom mantengono le informazioni necessarie a navigare, nel DAG, fino ai nodi ponte.

La tabella di routing ed i filtri di Bloom sono creati durante la fase di preprocessing, che consiste in più visite BFS. Durante ogni visita BFS, si costruisce una specifica parte della *tabella di routing* attraverso le funzioni *index.update* e *index.add* (vedi paragrafo 5.3) arrivando, alla fine del processo

ricorsivo, a recuperare tutte le informazioni necessarie al processo di routing (informazioni indispensabili per passare da un nodo ponte al successivo).

Poiché durante una visita BFS acquisiamo la conoscenza in modo incrementale (cioè a partire dal nodo radice, fino al raggiungimento di tutti i nodi), nelle funzioni $CreateCompassForNode$ e $CreateCompass$ (che si occupano di creare la tabella di routing e i filtri di Bloom) controlliamo, tramite la matrice delle distanze $dist$, se, percorrendo uno specifico arco, ci avviciniamo o meno ad un nodo. Tramite questi controlli (alle righe 5, 19, 29 e 36 della funzione $CreateCompassForNode$ e alle righe 18, 28, 35 e 40 della funzione $CreateCompass$)[2] ricaviamo le informazioni necessarie a creare la parte della tabella di routing per tutti i cammini minimi che utilizzano gli archi del DAG, percorrendoli in verso opposto alla visita BFS.

Durante la visita, si individuano i nodi ponte che vengono inseriti in tabella di routing, in corrispondenza di una specifica coppia orientata di nodi, sorgente-destinazione (si veda algoritmo 10 alla riga 6). Ogni nodo ponte b (a meno che non coincida con la destinazione finale) serve ad indicare, all'algoritmo 12, di decompressione dei cammini minimi, la necessità di continuare la visita in un diverso DAG, oppure la presenza di più direzioni possibili, per la destinazione finale.

In questo ultimo caso, il nodo ponte non ha solamente la funzione di collegare DAG diversi, ma anche quella di diminuire il numero di ponti presenti nella tabella di routing. Per questo motivo, nel caso in cui, in un DAG, un nodo abbia più archi che portano a nodi ponte diversi, per una stessa destinazione finale, tale nodo diventa un nodo ponte per quella destinazione (vedi figura 5.1).

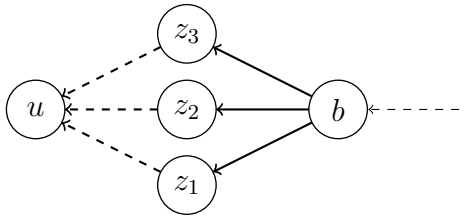


Figura 5.1: Ponte b che instrada in uno stesso DAG per il nodo destinazione u

[2]Notare che i controlli sono gli stessi descritti nel sottoparagrafo 4.2.2

Tale situazione accade se, e solo se, durante la fase di preprocessing, si individuano più ponti (direzioni) per una coppia di nodi destinazione-sorgente (y, n_1) (la nomenclatura si riferisce alla funzione $Jetsons.update$, algoritmo 9 nel paragrafo 5.3). In questo caso, il nodo y può diventare un ponte (per la destinazione n_1) per tutti i nodi x del DAG, i cui archi siano entranti in y. Questo però non accade, nel caso in cui, i nodi x abbiano strade alternative per n_1, cioè che utilizzino (tramite archi appartenenti ad altri DAG) cammini diversi, con costo minore di $d(y, n_1) + 1$. Infatti, la funzione $Jetsons.update$ è chiamata solo nel caso in cui la distanza di y da n_1 sia minore di 1 rispetto alla distanza tra x e lo stesso nodo n_1 (vedi algoritmi 7 e 8).

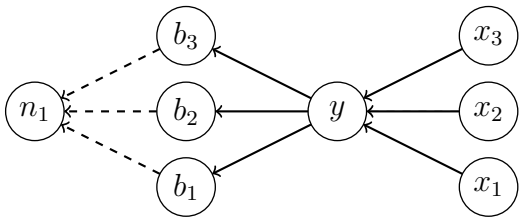


Figura 5.2: Il nodo y, poiché è connesso a ponti distinti (b_1, b_2 e b_3), diviene un nodo ponte per i nodi x_1, x_2 e x_3

La situazione sopra esposta, che descrive l'esistenza di più nodi ponte (in figura, denominati b_1, b_2 e b_3) per un dato nodo (in figura denominato n_1), corrisponde all'unico caso in cui è necessario riportare in tabella di routing più di un nodo ponte per una data coppia di vertici (chiave del dizionario). Ogni lista l dei nodi ponte, con lunghezza maggiore di uno, deriverà dall'unione di più liste $list$, tutte di lunghezza pari ad uno (riga 6 algoritmo 10 nel capitolo 5.3). Per questo motivo, considerando l'esempio in figura 5.2, interrogando la tabella tramite la chiave (y, n_1) otterremo la lista dei ponti $[b_1, b_2, b_3]$; mentre alle coppie (x_i, n_1) (con $1 \leq i \leq 3$) sarà associato il solo ponte y. Così procedendo, nella costruzione della tabella di routing, cerchiamo di ridurne il numero di elementi.

In tabella, uno stesso nodo ponte può essere inserito tante volte, quante sono le coppie per cui ricopre il ruolo di collegamento tra DAG, per la concatenazione di sottocammini, al fino di ottenere almeno un cammino minimo per la coppia data. Nel caso però, in cui, esistano più ponti per la coppia sorgente-destinazione, il processo di costruzione della tabella fa sì che, se possibile, venga individuato un nuovo nodo (nell'esempio, denominato y) che faccia da ponte per la coppia di nodi (nell'esempio (x_i, n_1)). Nello pseudocodice, tale operazione equivale a creare, durante la fase di preprocessing, una

lista di lunghezza uno $[y]$, contenente il solo nodo y, che faccia da ponte tra il nodo x e il nodo n_1(vedi riga 9 dell'algoritmo 9).

Durante le visite BFS della fase di preprocessing, utilizziamo una struttura temporanea, denominata *struttura dei Jetson*, al fine di mantenere la corrispondenza tra ogni futura direzione di ricerca (arco del DAG opposto alla BFS) e i nodi raggiungibili, tramite un cammino minimo. Questa struttura d'appoggio, al termine della visita, costruiti i filtri di Bloom e la porzione di tabella relativa, diverrà inutile, rendendo possibile la liberazione della memoria da essa occupata. In alternativa, per diminuire l'occupazione della memoria causata dalla struttura, è possibile eliminare le informazioni inutili, man mano che procediamo nella visita BFS; infatti, ad ogni passo, ci sono necessarie solamente le informazioni riguardanti gli archi appartenenti a due livelli [3] adiacenti del DAG: il livello che stiamo visitando ed il precedente (rispettivamente corrispondenti al nodo x e al nodo y degli algoritmi 7 e 8). La struttura dei Jetson consiste in un dizionario che ha, come chiavi, gli archi del DAG, e, come valori, coppie. Ad ogni arco (x, y) corrisponde una coppia, formata da un nodo n_1 (raggiungibile, nel grafo, tramite uno o più cammini minimi) e dalla lista l di ponti necessari a raggiungere detto nodo. L'associazione nel dizionario $Jetsons$, tra un arco (x, y), un nodo n_1 e dei ponti, elencati nella lista l, indica l'esistenza di almeno un cammino minimo (per ogni ponte) diretto al nodo, passante per l'arco. Tale associazione è effettuata tramite l'algoritmo 9.

Per meglio comprendere il ruolo, all'interno del processo di compressione, svolto dai nodi ponte insieme all'uso dei filtri di Bloom, è importante specificare che l'esistenza di un nodo ponte, per una data coppia di nodi sorgente-destinazione, non corrisponde all'esistenza di un unico cammino sorgente-destinazione. Infatti, i ponti sono determinati tramite un processo di visita BFS, durante il quale, gli stessi, se non perdono di validità (per la destinazione), sono ereditati da alcuni nodi del livello successivo. Per questo motivo, può succedere che, definiti x, y e b rispettivamente i due valori della tabella di routing e il nodo ponte risultante dall'interrogazione da essi specificata, esistano più cammini minimi, da x a b, i quali sono sottocammini di cammini minimi da x ad y (vedi figura 5.3).

Il compito di mantenere le informazioni (localmente ad ogni nodo) necessarie a ripercorrere tutti i cammini tra x e b, durante il processo di decompressione, spetta ai filtri di Bloom.

[3]Ogni livello del DAG è formato dai nodi equidistanti dal nodo (radice) da cui inizia la visita BFS e dai relativi archi uscenti

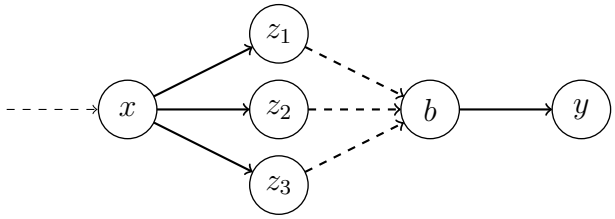


Figura 5.3: Esistenza di più cammini da x a b, passanti per z_1, z_2 e z_3

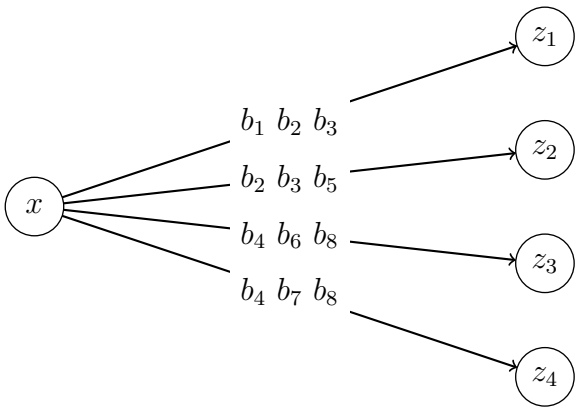
La struttura dei Jetson non è solamente utilizzata per costruire la tabella di routing, ma è anche la struttura alla base della creazione dei filtri di Bloom (vedi algoritmo 11).

Per ogni nodo di ogni DAG, è costruito un *Bloom filter* che serve per navigare nel DAG, a partire dal nodo al quale corrisponde. Il compito di ogni Bloom filter è quello di associare ad ogni arco del nodo (per cui è stato costruito) i nodi ponte raggiungibili, percorrendo tale arco[4].

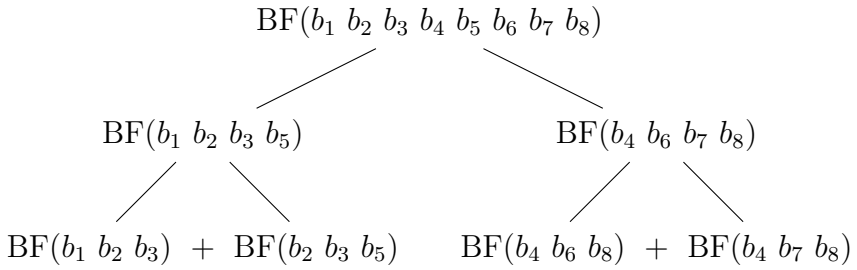
In realtà, ogni Bloom filter ha una struttura gerarchica ed è formato da più Bloom filter disposti ad albero binario (vedi esempio in figura 5.4). Disponiamo i filtri di Bloom su tale struttura, in modo da poter effettuare una ricerca binaria sull'albero. Per far ciò, costruiamo l'albero a partire dalle foglie e, man mano che saliamo di livello nell'albero, fondiamo coppie di nodi del livello precedente. Ogni foglia dell'albero corrisponde ad un arco del nodo (per cui si sta costruendo il filtro di Bloom) ed è costituita da un Bloom filter, rappresentante l'insieme dei nodi ponte, raggiungibili per quello specifico arco. Di conseguenza, per costruzione, ogni nodo interno dell'albero corrisponde ad un sottoinsieme di archi ed è formato da un Bloom filter, rappresentante l'insieme dei nodi ponte raggiungibili, percorrendo anche un solo arco di tale sottoinsieme (vedi figura 5.4). La costruzione dell'albero binario può essere effettuata cercando di minimizzare la probabilità di errore del Bloom filter gerarchico (si veda paragrafo degli sviluppi futuri 7.1).

La ricerca binaria è effettuata, nel momento in cui, interrogata la tabella di routing (algoritmo 12 riga 3), abbiamo un nodo x di partenza di un DAG ed un nodo ponte b da raggiungere. La distanza tra i due nodi può essere di uno, come di molteplici hop e corrispondere ad uno, o a molti cammini minimi. Per coprire tale distanza (ed individuare tutti i cammini minimi), si utilizzano uno, o più, filtri di Bloom (attraverso la funzione *edgesGen* definita nell'algoritmo 13 nel paragrafo 5.3), che, fungendo da bussole, ci indicano, ad ogni passo, le direzioni possibili. Per ogni coppia di nodi (partenza, ponte

[4]Si ricorda che, nel capitolo 4 nel sottoparagrafo 4.2.2, si utilizza la stessa struttura, riferendoci ad essa con l'espressione "bussola probabilistica".



(a) *Nodo x, su ogni arco sono riportati i ponti raggiungibili attraverso di esso.*



(b) *Bloom filter del nodo x. Ogni foglia, procedendo da sinistra a destra, mantiene l'insieme dei ponti raggiungibili rispettivamente attraverso gli archi (x, z_1), (x, z_2), (x, z_3), (x, z_4). I nodi interni mantengono gli insiemi dei ponti raggiungibili tramite gli archi corrispondenti alle foglie del loro sottoalbero.*

Figura 5.4: Bloom filter gerarchico del nodo x

di destinazione) interroghiamo tanti Bloom filter gerarchici, quanti sono gli hop da effettuare per percorrere tutti i cammini minimi tra i due nodi.

Ogni Bloom filter gerarchico è interrogato (riga 2 algoritmo 13) a partire dalla radice, ricercando nel Bloom filter ad essa corrispondente, il nodo ponte desiderato. In caso di risultato positivo, si procede ricorsivamente verso le foglie dell'albero. Così operando, riusciamo ad identificare il sottoinsieme di archi, che portano verso il nodo ponte considerato. Tale procedura ha costo (in termini di numero di controlli) proporzionale alla cardinalità dell'output.

Detto $\hat{E}$ il numero di archi del nodo corrispondente al Bloom filter gerarchico interrogato, al caso pessimo, si effettuano $\mathcal{O}(\log_2 \hat{E})$ controlli su altrettanti filtri di Bloom, al fine d'identificare l'unico arco che porta al ponte da raggiungere.

È importante notare che, per ogni nodo x di ogni DAG, esiste uno ed un solo filtro di Bloom $BF[x]$ (vedi algoritmo 13), che si occupa di instradare verso i ponti b (oggetto delle visite intermedie), che portano alla decompressione di tutti i cammini minimi, per una data coppia di nodi sorgente-destinazione.

5.1.4 Recupero dell'informazione

Costruite le strutture di dati ausiliarie, per l'individuazione di tutti i cammini minimi, per una qualsiasi coppia di vertici di un grafo dato G, è possibile rispondere ad una *path query*, recuperando tutti i cammini richiesti.

Data una coppia di nodi (s, t) per cui recuperare tutti i cammini minimi esistenti nel grafo G, il processo di recupero è effettuato visitando i DAG annidati, attraverso un processo ricorsivo, che andiamo di seguito a descrivere.

1. Interroghiamo la *tabella di routing*, tramite i parametri s e t, ottenendo una lista di coppie, ognuna delle quali, formate da:
   - un nodo, $\bar{s}$, appartenente ad uno specifico DAG, da cui iniziare la visita;
   - la relativa sequenza di nodi ponte, $[b_1, b_2, \ldots, b_k]$, da raggiungere, a partire da $\bar{s}$, per individuare alcuni sottocammini, che andranno a costituire la soluzione finale
2. Per ogni coppia $(\bar{s}, [b_1, b_2, \ldots, b_k])$
   - 2.1 Per ogni ponte b_i della lista (secondo elemento della coppia)
     - 2.1.1 Per ogni cammino minimo da $\bar{s}$ a b_i
       - (a) Generiamo tale cammino tramite la navigazione del *DAG*, usando i *filtri di Bloom*

(b) Se $(b_i \neq t)$

- effettuiamo la chiamata ricorsiva interrogando la tabella di routing per la coppia di nodi (b_i, t) e passando, come parametro, il sottocammino generato, da $\bar{s}$ a b_i

Altrimenti

- restituiamo un cammino minimo da s a t

Il processo di recupero dei cammini minimi, sopra descritto, consta di due procedure ricorsive (vedi algoritmi 12 e 13, sezione 5.3), che si chiamano a vicenda.

L'algoritmo di decompressione dei cammini, per una data coppia di nodi (s, t), recupera, tramite la tabella di routing, i nodi ponte e ne considera uno alla volta. Per ogni nodo ponte b, dopo aver percorso parte del DAG ed aver raggiunto la copia (nel DAG) di b, l'algoritmo controlla se tale copia corrisponde al nodo t del grafo, destinazione finale della ricerca. In caso negativo, continua ricorsivamente la ricerca, usando, come chiave della tabella, la coppia di nodi (b, t); in caso affermativo, l'algoritmo restituisce il cammino minimo individuato, con sorgente s e destinazione t. Qualora sia rimasto indietro un diverso nodo ponte b (non ancora utilizzato), la ricerca dei successivi cammini continua dirigendosi verso detto nodo ponte (anch'esso precedentemente individuato, tramite una richiesta alla tabella di routing). Il compito di recuperare i cammini minimi da un nodo ponte all'altro è affidato alla procedura 13, che naviga in un DAG attraverso i filtri di Bloom. In questo modo, recuperando e concatenando i diversi sottocammini dei cammini minimi, riusciamo ad individuare tutti i cammini minimi dalla sorgente s alla destinazione t.

5.2 Esempi di applicazione

Nel seguente paragrafo, per meglio comprendere il funzionamento della soluzione sopra esposta, riportiamo due esempi di applicazione della stessa a due diversi grafi. In questo modo, riusciamo ad aggiungere dettagli importanti, riguardanti le strutture e gli algoritmi di creazione delle stesse, dettagli non inseriti nel paragrafo precedente, per non distrarre dalla struttura generale della soluzione.

Nel paragrafo successivo, tramite pseudocodice saranno formalmente definite le strutture di dati e gli algoritmi di risoluzione.

Il primo esempio, qui riportato, ha il compito di mostrare le strutture create durante la fase di preprocessing: DAG, tabella di routing e filtri

di Bloom. Il secondo è utile per comprendere la complessità alla base del problema che ci siamo posti di risolvere, e la conseguente complessità delle strutture di annidamento che si vengono a creare.

5.2.1 Primo esempio

Per poter meglio comprendere la procedura sopra descritta, la applichiamo al grafo campione (grafo in figura 3.1), utilizzato per gli esempi effettuati nel capitolo 3, grafo, che riportiamo qui sotto per facilitare la leggibilità e la comprensione dei risultati dell'applicazione.

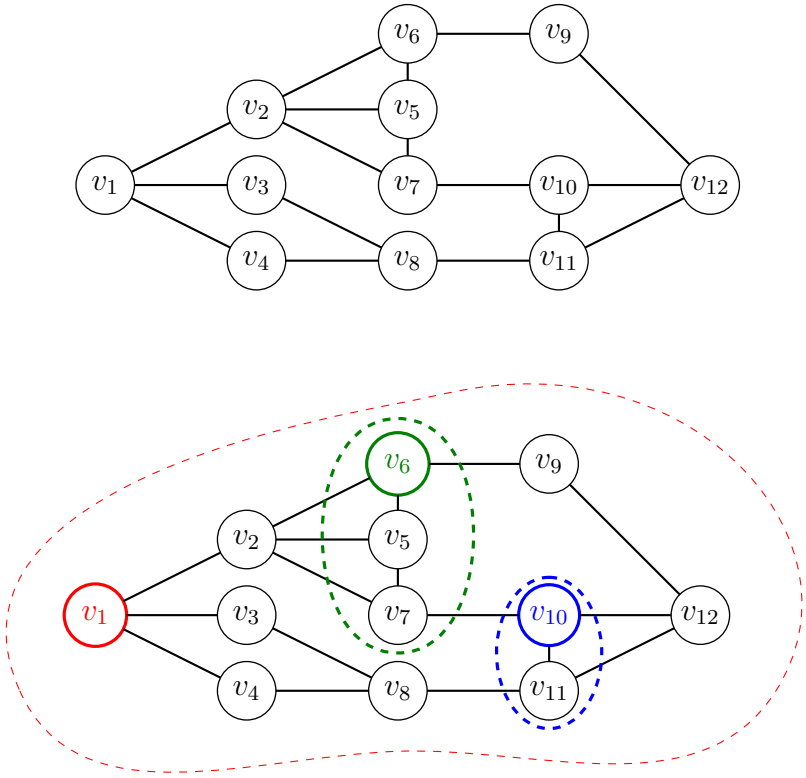


Figura 5.5: Partizionamento del grafo in figura 3.1, in tre parti

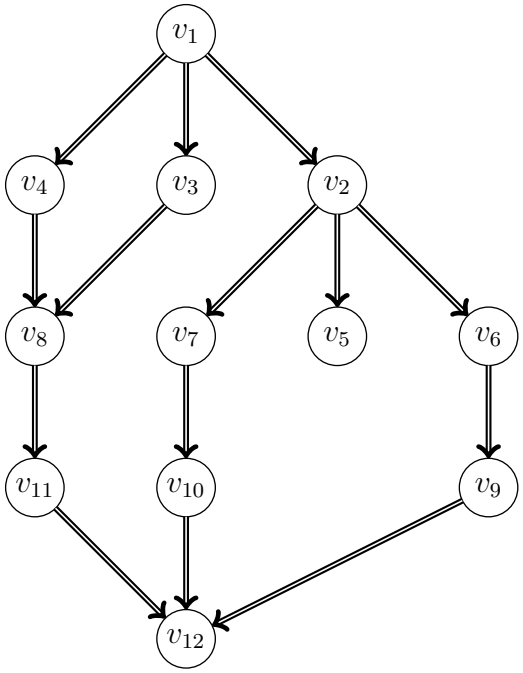
La soluzione consiste di una prima fase di preprocessing, in cui vengono creati i DAG (algoritmi 5 e 6), la tabella di routing (algoritmo 10) e i filtri di Bloom (algoritmo 11), ed una successiva fase d'interrogazione delle strutture, durante la quale si risponde a una richiesta (path query), recuperando tutti i cammini minimi per una coppia di nodi specificata (algoritmi 12 e 13).

La fase di preprocessing per il grafo sottostante consiste di tre visite BFS (su diverse porzioni di grafo) e di altrettante visite speculari. La prima visita è effettuata, considerando tutto il grafo, a partire dal nodo v_1, poiché (insieme al nodo v_{12}) v_1 è uno dei nodi con diametro massimo. Le successive visite sono fatte sui sottoinsiemi di nodi, formati dai nodi equidistanti da v_1 (in figura raggruppati tramite due ellissi di colore verde e blu). La seconda visita inizia dal nodo v_6 e tocca i nodi v_5, v_6 e v_7, mentre la terza considera solamente i vertici v_{10} e v_{11}.

Durante le tre visite BFS (a partire dalle tre "radici", v_1, v_6 e v_{10}), costruiamo i sei DAG riportati nelle figure 5.6, 5.7, 5.8, 5.9, 5.10 e 5.11. Sempre durante le stesse visite, vengono creati i filtri di Bloom ed i nodi ponte (inseriti nella tabella di routing), per i DAG "speculari", cioè i DAG i cui archi hanno senso opposto alla direzione di visita. Nell'esempio sottostante, i DAG "speculari" sono quelli riportati nelle figure 5.9, 5.10 e 5.11.

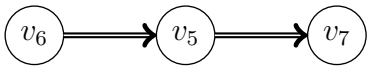
Dopo ogni visita BFS, è necessario operare un'ulteriore visita in ampiezza, ma in senso opposto (visita "speculare"). Tale visita è facilmente effettuata visitando i DAG "speculari", creati al passo precedente (algoritmo 8). La necessità di ripercorrere i nodi del DAG, ma in senso opposto, è data dall'esigenza di individuare i nodi ponte e costruire i filtri di Bloom, per il DAG, che ha gli archi con verso concorde con quello della visita BFS. Gli algoritmi *DAGs* e *DAGs_FROM* (numero 5 e 6) portano alla creazione di due DAG "speculari" che permettono di visitare il grafo in una stessa direzione, ma in due versi opposti.

Sviluppando l'esempio, in particolare rappresentando i DAG, viene naturale chiedersi se tali strutture possano essere tra loro combinate, in modo da comprimerle. A una prima analisi effettuata, risulta possibile combinare tra loro i diversi DAG ma, se da una parte tale operazione elimina alcuni ponti esistenti tra i DAG, dall'altra, favorisce la nascita di nuovi. Infatti, l'unione di più DAG porta alla fusione delle copie dei nodi corrispondenti allo stesso nodo del grafo; questo fa sì che aumenti il grado dei nodi dei DAG e si creino situazioni, come quella descritta in figura 5.2, che portano alla creazione di nuovi ponti. Per questo motivo, non è banale risolvere il problema di comprimere le strutture, fondendo i DAG; perciò rimandiamo la sua trattazione ad una fase successiva di sviluppo (vedi paragrafo 7.1).



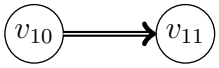
$\boldsymbol{v_1}$,$\boldsymbol{v_2}$,$\boldsymbol{v_6}$,$\boldsymbol{v_9}$,$\boldsymbol{v_{12}}$; $\boldsymbol{v_1}$,$\boldsymbol{v_2}$,$\boldsymbol{v_7}$,$\boldsymbol{v_{10}}$,$\boldsymbol{v_{12}}$; $\boldsymbol{v_1}$,$\boldsymbol{v_3}$,$\boldsymbol{v_8}$,$\boldsymbol{v_{11}}$,$\boldsymbol{v_{12}}$; $\boldsymbol{v_1}$,$\boldsymbol{v_4}$,$\boldsymbol{v_8}$,$\boldsymbol{v_{11}}$,$\boldsymbol{v_{12}}$;
v_2, v_6, v_9, v_{12} ; v_2, v_7, v_{10}, v_{12} ; v_3, v_8, v_{11}, v_{12} ; v_4, v_8, v_{11}, v_{12} ;
v_6, v_9, v_{12} ; v_7, v_{10}, v_{12} ; v_8, v_{11}, v_{12} ; v_8, v_{11}, v_{12} ;
v_9, v_{12} ; v_{10}, v_{12} ; v_{11}, v_{12} ; v_{11}, v_{12} ;

Figura 5.6: DAG generato da una visita BFS del grafo in figura 3.1 a partire dal nodo v_1. Sotto il DAG, si riportano anche tutti i cammini minimi da v_1 a v_{12} e i loro relativi sottocammini terminanti in v_{12} relativi al grafo in figura 3.1, così da osservare la compressione dei cammini minimi effettuata utilizzando il DAG.



$\boldsymbol{v_6}$,$\boldsymbol{v_5}$,$\boldsymbol{v_7}$; v_6,v_5 ; v_5,v_7 ;

Figura 5.7: DAG, annidato al DAG in figura 5.6, generato da una visita BFS operata a partire da v_6 e relativi cammini minimi individuati



$\boldsymbol{v_{10}}$,$\boldsymbol{v_{11}}$;

Figura 5.8: DAG, annidato al DAG in figura 5.6, generato da una visita BFS operata a partire da v_{10} e relativo cammino minimo individuato

La creazione dei DAG "speculari", qui sotto riportati, serve per poter individuare tutti i cammini minimi esistenti, anche quelli che iniziano "percorrendo il grafo in una direzione con uno specifico verso" (usando uno specifico DAG), per poi "andare nel verso opposto" (usando il DAG ad esso speculare). Per esempio, il cammino minimo del grafo che stiamo considerando, dato dalla sequenza di nodi v_6, v_2, v_7, è formato da archi del DAG in figura 5.6 e del relativo DAG speculare in figura 5.9.

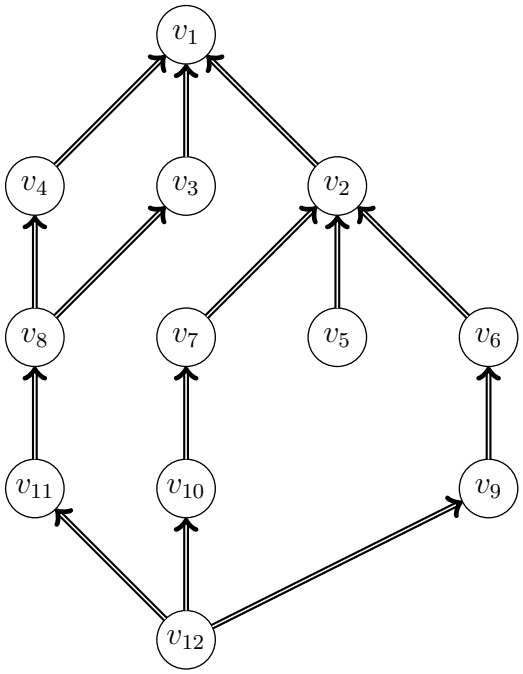


Figura 5.9: DAG in verso opposto rispetto al DAG in figura 5.6

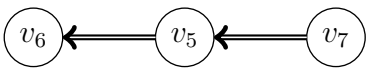


Figura 5.10: DAG in verso opposto rispetto al DAG in figura 5.7

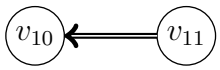


Figura 5.11: DAG in verso opposto rispetto al DAG in figura 5.8

L'applicazione della procedura di preprocessing partiziona, quindi, il grafo in tre coppie di DAG, formate da DAG tra loro speculari. Per poter recuperare tutti i cammini minimi per tutte le coppie di vertici è necessario mantenere i DAG (riportati nelle figure), i filtri di Bloom relativi a tutti i nodi di tutti i vertici dei DAG e la tabella di routing.

A pagina seguente riportiamo la tabella di routing (tabella 5.1), così da mostrare i ponti esistenti tra i vari DAG. Ogni cella nella tabella, corrispondente alla riga x ed alla colonna y, contiene lo copie di x (relative ai diversi DAG) da cui iniziare la visita, per il recupero dei cammini minimi per la coppia di vertici (x, y) ed i relativi nodi ponte da utilizzare. In tabella, il colore e lo stile dei nodi indica l'appartenenza del nodo ad uno specifico DAG. I colori usati sono in coerenza con la figura 5.5, che partiziona il grafo in base ai DAG creati, mentre lo stile MAIUSCOLETTO indica l'appartenenza del nodo ad un DAG "speculare".

Tabella 5.1: Tabella di routing riportante i "nodi ponte"

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}
v_1	v_1,v_1	v_1,v_2	v_1,v_3	v_1,v_4	v_1,v_5	v_1,v_6	v_1,v_7	v_1,v_8	v_1,v_9	v_1,v_{10}	v_1,v_{11}	v_1,v_{12}
v_2	V$_2$,V$_1$	v_2,v_2	V$_2$,V$_1$	V$_2$,V$_1$	v_2,v_5	v_2,v_6	v_2,v_7	V$_2$,V$_1$	v_2,v_9	v_2,v_{10}	v_2,v_7	v_2,v_{12}
v_3	V$_3$,V$_1$	V$_3$,V$_1$	v_3,v_3	V$_3$,V$_1$ $\boldsymbol{v_3}$,$\boldsymbol{v_8}$	V$_3$,V$_1$	V$_3$,V$_1$	V$_3$,V$_1$	v_3,v_8	V$_3$,V$_1$	$\boldsymbol{v_3}$,$\boldsymbol{v_{11}}$	v_3,v_{11}	v_3,v_{12}
v_4	V$_4$,V$_1$	V$_4$,V$_1$	V$_4$,V$_1$ $\boldsymbol{v_4}$,$\boldsymbol{v_8}$	v_4,v_4	V$_4$,V$_1$	V$_4$,V$_1$	V$_4$,V$_1$	v_4,v_8	V$_4$,V$_1$	$\boldsymbol{v_4}$,$\boldsymbol{v_{11}}$	v_4,v_{11}	v_4,v_{12}
v_5	V$_5$,V$_1$	V$_5$,V$_2$	V$_5$,V$_1$	V$_5$,V$_1$	v_5,v_5	V$_5$,V$_6$	v_5,v_7	$\boldsymbol{v_5}$,$\boldsymbol{v_7}$ V$_5$,V$_1$	V$_5$,V$_6$	$\boldsymbol{v_5}$,$\boldsymbol{v_7}$	$\boldsymbol{v_5}$,$\boldsymbol{v_7}$	V$_5$,V$_6$ $\boldsymbol{v_5}$,$\boldsymbol{v_7}$
v_6	V$_6$,V$_1$	V$_6$,V$_2$	V$_6$,V$_1$	V$_6$,V$_1$	v_6,v_5	v_6,v_6	V$_6$,V$_2$ v_6,v_7	V$_6$,V$_1$ $\boldsymbol{v_6}$,$\boldsymbol{v_{12}}$	v_6,v_9	V$_6$,V$_2$ $\boldsymbol{v_6}$,$\boldsymbol{v_{12}}$ $\boldsymbol{v_6}$,$\boldsymbol{v_7}$	$\boldsymbol{v_6}$,$\boldsymbol{v_{12}}$	v_6,v_{12}
v_7	V$_7$,V$_1$	V$_7$,V$_2$	V$_7$,V$_1$	V$_7$,V$_1$	V$_7$,V$_5$	V$_7$,V$_2$ V$_7$,V$_5$	v_7,v_7	$\boldsymbol{v_7}$,$\boldsymbol{v_{10}}$	V$_7$,V$_2$ V$_7$,V$_6$ $\boldsymbol{v_7}$,$\boldsymbol{v_{12}}$	v_7,v_{10}	$\boldsymbol{v_7}$,$\boldsymbol{v_{10}}$	v_7,v_{12}
v_8	V$_8$,V$_1$	V$_8$,V$_1$	V$_8$,V$_3$	V$_8$,V$_4$	V$_8$,V$_1$ $\boldsymbol{v_8}$,$\boldsymbol{v_{11}}$	V$_8$,V$_1$ $\boldsymbol{v_8}$,$\boldsymbol{v_{12}}$	$\boldsymbol{v_8}$,$\boldsymbol{v_{11}}$	v_8,v_8	$\boldsymbol{v_8}$,$\boldsymbol{v_{12}}$	$\boldsymbol{v_8}$,$\boldsymbol{v_{11}}$	v_8,v_{11}	v_8,v_{12}
v_9	V$_9$,V$_1$	V$_9$,V$_2$	V$_9$,V$_1$ $\boldsymbol{v_9}$,$\boldsymbol{v_{12}}$	V$_9$,V$_1$ $\boldsymbol{v_9}$,$\boldsymbol{v_{12}}$	V$_9$,V$_6$[1]	v_6,v_9	V$_9$,V$_6$[1] V$_9$,$\boldsymbol{v_{12}}$	$\boldsymbol{v_9}$,$\boldsymbol{v_{12}}$	v_9,v_9	$\boldsymbol{v_9}$,$\boldsymbol{v_{12}}$	$\boldsymbol{v_9}$,$\boldsymbol{v_{12}}$	v_9,v_{12}
v_{10}	V$_{10}$,V$_1$	V$_{10}$,V$_2$	$\boldsymbol{v_{10}}$,$\boldsymbol{v_{11}}$	$\boldsymbol{v_{10}}$,$\boldsymbol{v_{11}}$	V$_{10}$,V$_7$	V$_{10}$,V$_7$ $\boldsymbol{v_{10}}$,$\boldsymbol{v_{12}}$	V$_{10}$,V$_7$	$\boldsymbol{v_{10}}$,$\boldsymbol{v_{11}}$	$\boldsymbol{v_{10}}$,$\boldsymbol{v_{12}}$	v_{10},v_{10}	v_{10},v_{11}	v_{10},v_{12}
v_{11}	V$_{11}$,V$_1$	V$_{11}$,V$_{10}$	V$_{11}$,V$_3$	V$_{11}$,V$_4$	V$_{11}$,V$_{10}$	$\boldsymbol{v_{11}}$,$\boldsymbol{v_{12}}$	V$_{11}$,V$_{10}$	V$_{11}$,V$_8$	$\boldsymbol{v_{11}}$,$\boldsymbol{v_{12}}$	V$_{11}$,V$_{10}$	v_{11},v_{11}	v_{11},v_{12}
v_{12}	V$_{12}$,V$_1$	V$_{12}$,V$_2$	V$_{12}$,V$_3$	V$_{12}$,V$_4$	V$_{12}$,V$_5$	V$_{12}$,V$_6$	V$_{12}$,V$_7$	V$_{12}$,V$_8$	V$_{12}$,V$_9$	V$_{12}$,V$_{10}$	V$_{12}$,V$_{11}$	v_{12},v_{12}

[1]Si considera solamente il nodo ponte v_6, perché questo può generare i cammini minimi anche per il ponte v_2.

5.2.2 Secondo esempio

Il seguente esempio ha l'utilità di mostrare come anche un semplice grafo possa avere una struttura di connessione, tale da rendere necessario l'utilizzo di più DAG tra loro annidati.

Considerato il grafo in figura 5.12, riportiamo l'albero di ricorsione (figura 5.13) e i DAG costruiti in fase di preprocessing (figure 5.14, 5.15 e 5.16), evitando di rappresentare i DAG "speculari", perché isomorfi a quelli rappresentati.

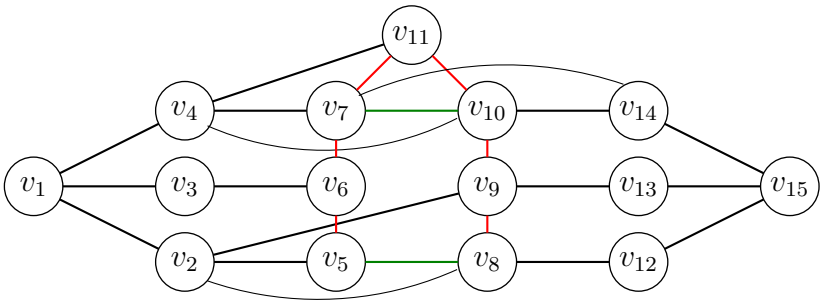


Figura 5.12: Grafo di esempio, con evidenziati i diversi DAG creati durante la fase di preprocessing

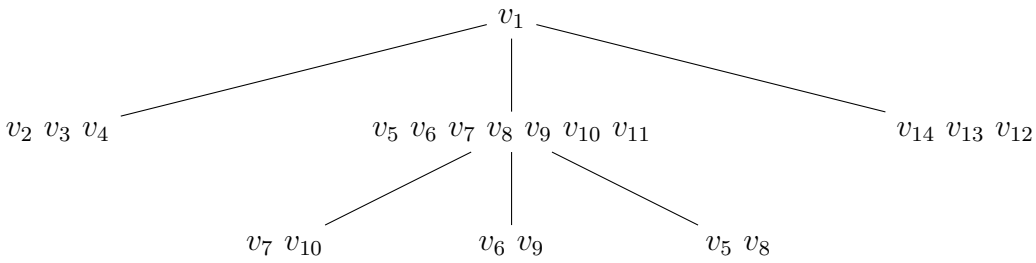


Figura 5.13: Albero di ricorsione rappresentante i sottoinsiemi su cui viene effettuata la ricorsione

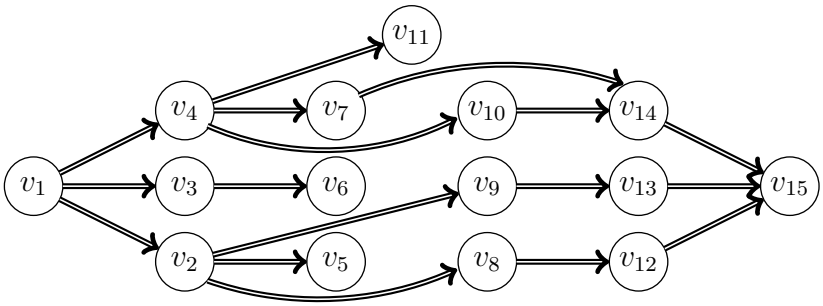


Figura 5.14: DAG1

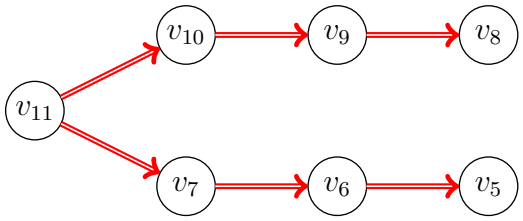


Figura 5.15: DAG2

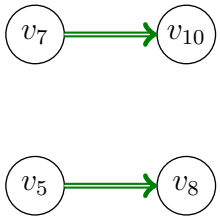


Figura 5.16: DAG3 e DAG4

5.3 Algoritmi per la rappresentazione compatta di cammini minimi

Di seguito riportiamo lo pseudocodice della procedura di recupero e memorizzazione di tutti i cammini minimi in un grafo non diretto.

Algoritmo 2 Creazione di tutti i DAG usati per la memorizzazione di tutti i cammini minimi presenti in un grafo G

```
1: procedure DAGs_CREATION(G, dist)   ▷ Grafo e matrice delle dist
2:     n ← |V|
3:     m ← |E|
4:     DAGs = {}
5:     s ← choose(G, dist)                          ▷ Scelta della radice
6:     DAGs(s, index, dist, DAGs)        ▷ Costruzione ricorsiva dei DAG
7:     return DAGs
8: end procedure
```

Algoritmo 3 Algoritmo di scelta del primo nodo radice

```
procedure CHOOSE(G, dist)                    ▷ Grafo, matrice delle dist
...          ▷ Scelta n ∈ V : ∃x ∈ V : dist[n][x] ≥ dist[n1][y]   ∀n1, y ∈ V
...
end procedure
```

Algoritmo 4 Algoritmo di scelta di un nodo radice in Peers

```
procedure CHOOSE(P, dist)     ▷ Insieme di nodi Peers, matr. dist
...          ▷ Scelta n ∈ P : ∃x ∈ P : dist[n][x] ≥ dist[n1][y]   ∀n1, y ∈ P
... P = P \ {n}                                          ▷ Modifico P
end procedure
```

Algoritmo 5 Algoritmo per la creazione del DAG radicato in s (alg. con alla base l'alg.1 per la generazione in ordine BFS dei nodi di un grafo)

```
    procedure DAGs(s, index, dist, DAGs)
        DAG1 ← new_DAG(s)                          ▷ DAG ''radicato'' in s
        DAG2 ← new_DAG(s)                  ▷ DAG (speculare) che ha s come
    ''target''
        Q ← new_queue()                                  ▷ Coda visita BFS
 5:     Q.push(s)
        M = {s}                                  ▷ Insieme dei nodi visitati
        Peers = {}                               ▷ Nodi ad uno stesso livello
        Jetsons = dict()                 ▷ var appoggio per costruzione bussola
        Wells ← {}                       ▷ Pila di nodi pozzo (radici speculari)
10:     while Q is not empty do
            v ← Q.pop()
            if v ≠ s and dist[s][v'] ≠ dist[s][v] then   ▷ Nuovo livello di nodi
                M = M ∪ Peers                  ▷ Marco i nodi visti fino ad ora
                DAGs = DAGs ∪ DAGs_FROM(Peers, index, dist, DAGs
15:             Peers = {}                          ▷ Svuoto l'insieme dei Peers
            end if
            for all c ∈ v.neighbors() do
                if c is not in M then
                    if c is not in Peers then              ▷ Costruzione dei DAG
20:                     Peers.add(c)
                        Q.push(c)
                        DAG1.addNode(c)
                        DAG1.addEdge(v, c)
                        DAG2.addNode(c)
25:                     DAG2.addEdge(c, v)
                    else
                        DAG1.addEdge(v, c)
                        DAG1.addEdge(c, v)
                    end if
30:             end if
            end for
            createCompassForNode(Jetsons, s, v, dist, DAG2, index)
            if ∄c ∈ v.neighbors() : c ∉ M then
                Wells = Wells.push(v, dist[s][v])                ▷ v è un pozzo
35:         end if
            v' ← v                                       ▷ Variabile d'appoggio
        end while
        createCompass(Wells, dist, DAG1, DAG2, index)
        DAGs = DAGs ∪ DAG1, DAG2
40:     return DAGs
    end procedure
```

Algoritmo 6 Algoritmo per la creazione ricorsiva del DAG (variante dell'alg.5)

```
    procedure DAGs_FROM(Nodes, edges, index, dist,DAGs)
        while Nodes is not empty do
            s ← choose(Nodes, dist)                          ▷ Estrae un nodo
            DAG1 ← new_DAG(s)
 5:         DAG2 ← new_DAG(s)
            Q ← new_queue()
            Q.push(s)
            M = {s}
            Peers = {}
10:         Jetsons = dict()
            while Q is not empty do
                v ← Q.pop()
                if v ≠ s and dist[s][v'] ≠ dist[s][v] then
                    M = M ∪ Peers
15:                 DAGs = DAGs ∪ DAGs_FROM(Peers, index, dist, DAGs)
                    Peers = {}
                end if
                for all c ∈ v.neighbors() do
                    if c is not in M then
20:                     if c is in Nodes then
                            if c is not in Peers then
                                Peers.add(c)
                                Q.push(c)
                                DAG1.addNode(c)
25:                             DAG1.addEdge(v, c)
                                DAG2.addNode(c)
                                DAG2.addEdge(c, v)
                            else
                                DAG1.addEdge(v, c)
30:                             DAG2.addEdge(c, v)
                            end if
                            Nodes = Nodes \ {c}
                        end if
                    end if
35:             end for
                createCompassForNode(Jetsons, s, v, dist, DAG2, index)
                if ∄c ∈ v.neighbors() : c ∉ M then
                    Wells = Wells ∪ (v, dist[s][v])
                end if
40:             v' ← v
                createBF(v, DAG1)
            end while
        end while
        createCompass(Wells, dist, DAG1, DAG2, index)
45:     DAGs = DAGs ∪ DAG1, DAG2
        return DAGs
    end procedure
```

Algoritmo 7 Algoritmo che crea la tabella di routing e Bloom filter per navigare il DAG dalla sorgente fino ai pozzi

```
procedure createCompassForNode(Jetsons,s,x,dist,DAG2,index)
    if x == s then                                  ▷ Se x è la radice del DAG
        for all (s, y) ∈ s.edges() do                                  ▷ Creo
            for all n ∈ V such that (y, s) ∉ DAG2 do
                if dist[s][n] − dist[y][n] == 1 then
                    Jetsons[(y, s)].add(newValue(n, [s]))   ▷ Link tra DAGs
                end if
            end for
        end for
    else                                                  ▷ Eredito i figli
        for all (x, y) ∈ x.edges() do              ▷ Per ogni arco nel grafo
            if (x, y) ∈ DAG2.edges(x) then              ▷ (x, y) uscente da x
                Jetsons[(x, y)].add(newValue(y, [y]))
                index.update(x, y, [y])
                for all (y, u) ∈ DAG2.edges(y) do
                    setValue ← (setValue ∪ Jetsons[(y, u)])
                end for
                for all (n1, l) ∈ setValue do              ▷ se i nipoti di y
                    if dist[x][n1] − dist[y][n1] == 1 then          ▷ sono tali
                        Jetsons.update((x, y), n1, l)               ▷ anche per x
                        index.update(x, n1, l)
                    end if                              ▷ Tengo traccia dei nipoti
                end for                                    ▷ e dei relativi ponti
                setValue ← {}
                for all (y, u) ∉ DAG2.edges(y) such that u ≠ x do
                    setValue ← (setValue ∪ Jetsons[(y, u)])
                end for
                for all (n2, l) ∈ setValue do              ▷ se i nipoti di y
                    if dist[x][n2] − dist[y][n2] == 1 then          ▷ sono tali
                        Jetsons.update((x, y), n2, [y])             ▷ anche per x
                        index.update(x, n2, [y])
                    end if                              ▷ Tengo traccia dei nipoti
                end for                                    ▷ e dei relativi ponti
            else
                for all n ∈ V do
                    if dist[x][n] − dist[y][n] == 1 then
                        Jetsons.update((x, y), n, [x])          ▷ Link tra DAGs
                    end if
                end for
            end if
        end for
        createBF(x, Jetsons, DAG2)
    end if
end procedure
```

Algoritmo 8 Algoritmo che crea la bussola per navigare il DAG1 (variante dell'alg.7)

```
procedure createCompass(Wells,dist,DAG1,DAG2,index)
    Jetsons = dict(); nWells = 1; nPeers = 0
    dist ← Wells.top().d; w ← Wells.pop().v
    Q ← new_queue(); Q.push(w); M = {w}                    ▷ Visita BFS
    while Q is not empty do          ▷ Rinavigo il DAG2 costruito in BFS
        if nWells == 0 then
            nWells = nPeers; nPeers = 0; dist − −
        while Wells.top().d == dist do
            Q.push(Wells.pop().v); nWells + +
        end while
        x ← Q.pop(); nWells − −
        for all c ∈ DAG2.edges(x) do
            if c is not in M then
                M.add(c); Q.push(c); nPeers + +
                                ▷ Man mano che avanzo costruisco la bussola
    if x ∈ DAG2.source() then          ▷ Se x è una sorgente di DAG2
        for all (x, y) ∈ x.edges() do                  ▷ Per ogni arco del grafo
            for all n ∈ V do
                if dist[x][n] − dist[y][n] == 1 then
                    Jetsons[(x, y)].add(newValue(n, [y]))
    else                                                    ▷ Eredito i nipoti
        for all (x, y) ∈ x.edges() do                      ▷ Per ogni arco di x
            if (x, y) ∈ DAG1.edge(x) then                   ▷ arco uscente da x
                Jetsons[(x, y)].add(newValue(y, [y]))
                index.update(x, y, [y])
                for all (y, u) ∈ DAG1.edges(y) do
                    setValue ← (setValue ∪ Jetsons[(y, u)])
                for all (n1, l) ∈ setValue do
                    if dist[x][n1] − dist[y][n1] == 1 then
                        Jetsons.update((x, y), n1, l)
                        index.update(x, n1, l)
                                                ▷ Tengo traccia dei nipoti
                                                  ▷ e dei relativi ponti l
                setValue ← {}
                for all (y, u) ∉ DAG1.edges(y) such that u ≠ x do
                    setValue ← (setValue ∪ Jetsons[(y, u)])
                for all (n2, l) ∈ setValue do
                    if dist[x][n2] − dist[y][n2] == 1 then
                        Jetsons.update((x, y), n2, [y])▷ Tengo traccia dei nipoti
                        index.update(x, n2, [y])          ▷ e dei relativi ponti
            else
                for all n ∈ V do
                    if dist[x][n] − dist[y][n] == 1 then
                        Jetsons.update((x, y), n, [x])
        createBF(x, Jetsons, DAG1)
    end while
end procedure
```

Algoritmo 9 Algoritmo di aggiornamento del dizionario *Jetsons*

```
procedure JETSONS.UPDATE((x, y), n_1, list)      ▷ Arco,nipote,lista ponti
    setValue ← Jetsons[(x, y)]
    found ← false
    while found == false do
        for all (n_2, l) ∈ setValue do
            if n_1 == n_2 then
                l = (l ∪ list)                   ▷ Aggiorno la lista dei ponti
                if |l| > 1 then
                    l = [y]                      ▷ Creo un nuovo ponte (figura 5.2)
                end if
                found ← true
                break
            end if
        end for
    end while
    if found == false then
        Jetsons[(x, y)].add(newValue(n_1, list))
    end if
end procedure
```

Algoritmo 10 Algoritmo per l'aggiornamento della tabella di routing

```
procedure INXED.UPDATE(x,t,list)     ▷ Nodo nel DAG, destinazione,
    found ← false                                  ▷ lista nodi ponte
    while found == false do
        for all (s, l) ∈ index[(x, t)] do
            if s == x then
                l = (l ∪ list)                   ▷ Aggiorno la lista dei ponti
                found ← true
                break
            end if
        end for
    end while
    if found == false then
        index[(x, t)].add(newValue(t, list))
    end if
end procedure
```

Algoritmo 11 Creazione del Bloom Filter per il nodo v

```
procedure createBF(v,Jetsons,DAG)                    ▷ nodo, nipoti, DAG
    for all e ∈ DAG.edges(v) do
        setNodes ← (setNodes ∪ Jetsons[e])
    end for
...     ▷ Crea Bloom Filter gerarchico del nodo v per |setNodes| elementi
...                 ▷ tale BF sarà costituito da log|DAG.edges(v)| livelli
...   ▷ Creo il BF cercando dei far discendere da uno stesso nodo il BF
simili
...                       ▷ Associo il BF creato al relativo nodo del DAG
end procedure
```

Le seguenti procedure (algoritmi 12 e 13) sono utilizzate per recuperare tutti i cammini minimi per una data coppia di nodi. La chiamata per effettuare una *path query*, su una coppia di nodi del grafo G (per il quale sono state costruite le strutture dati di appoggio) consiste nella chiamata di funzione: $shortestPaths((x, y), \epsilon)$, dove ϵ equivale al cammino vuoto (di lunghezza pari a zero).

Queste due procedure si chiamo a vicenda: la prima (algoritmo 12), trovati i ponti per una coppia di nodi (sorgente-destinazione), per ogni ponte, chiama la seconda per individuare un cammino dalla sorgente al ponte, indicandole il nodo destinazione; la seconda (algoritmo 13), recupera un cammino dalla destinazione al ponte ("chiamandosi" più volte) e, successivamente, chiama la prima procedura per recuperare le successive porzioni dei cammini ponte-destinazione.

Algoritmo 12 Algoritmo di recupero di tutti i cammini minimi

```
    procedure shortestPaths((x, y), path)          ▷ Sorgente e target
2:      path1 ← path.x                             ▷ Concateno path e x
        for all (s, listB) ∈ index[(x, y)] do                  ▷ Routing
4:          for all b ∈ listB do                   ▷ Ponti per raggiungere y
                edgesGen(x, b, y, path1)   ▷ Seq. di nodi per raggiungere b
6:          end for
        end for
8:  end procedure
```

Algoritmo 13 Generatore degli archi verso una data destinazione

```
procedure EDGESGEN((x, b, y, path))                ▷ Sorgente e target
    for all (x, z) ∈ BF[x].to(b) do
        if z ≠ b then                              ▷ Navigo sullo stesso DAG
            path1 = path.z                         ▷ Concateno path e z
            edgesGen(z, b, y, path1)
        else
            if b == y then
                return path.b                      ▷ Sono arrivato al target
            else
                shortestPaths(b, y, path)          ▷ Continuo su un altro DAG
            end if
        end if
    end for
end procedure
```

Capitolo 6

Analisi dei costi

L'obiettivo che ci siamo posti all'inizio della ricerca, qui descritta, non è tanto quello di determinare la soluzione ottima al problema di rappresentare tutti i cammini minimi di un grafo non diretto e non pesato, quanto di studiare in che quantità questi cammini possano essere tra loro simili e quali siano gli elementi strutturali dei grafi, che portano ad avere tali similarità. Tutto ciò, al fine ultimo di individuare una strategia risolutiva che riesca ad attaccare un problema così complesso, riuscendo a risolverlo con risultati apprezzabili.

Nel seguente paragrafo, riportiamo l'analisi dei costi della soluzione proposta (vedi capitolo 5), descrivendo i costi dati dalla costruzione delle strutture create in fase di preprocessing e quelli derivanti dall'utilizzo e dall'occupazione di memoria delle stesse.

6.1 Fase di preprocessing

La fase precedente alla richiesta di recupero di tutti i cammini minimi per una data coppia di nodi di un grafo, G, consiste nella costruzione dei DAG, della tabella di routing e dei filtri di Bloom. Assumiamo di conoscere solamente il grafo G. Prima di iniziare a visitare tale grafo e, man mano, che procediamo nella visita, per creare le strutture necessarie in fase di interrogazione, è essenziale acquisire la matrice delle distanze di G.

Il recupero della matrice delle distanze può essere effettuato tramite n visite in ampiezza, ognuna delle quali eseguita a partire dagli n nodi del grafo. Il costo di ogni visita è di $\mathcal{O}(n + m)$, per un complessivo costo, delle visite, di $\mathcal{O}(n \cdot m)$ passi.

Ottenuta la matrice delle distanze, la vera e propria fase di preprocessing, consiste di più visite BFS di porzioni diverse del grafo. Durante tali visite, vengono costruiti i DAG (strutture di compressione dei cammini minimi). Per

questo motivo il costo computazionale delle visite è strettamente legato al costo di occupazione della memoria dei DAG. Rimandiamo, perciò, il calcolo del costo delle visite in ampiezza, al sottoparagrafo che analizza l'occupazione in memoria dei DAG e delle altre strutture ausiliarie alla decompressione (vedi sottoparagrafo 6.2). Per adesso, assumiamo il costo di tali visite pari a $\mathcal{O}(n \cdot m)$.

Oltre alla costruzione dei DAG, la fase di preprocessing si occupa di creare la tabella di routing e i filtri di Bloom. Il costo della loro creazione è fortemente legato alla struttura del grafo G e dei suoi cammini minimi. La procedura, che si occupa della creazione della tabella e dei filtri di Bloom, cerca di diminuire il costo computazionale, mantenendo in memoria tutte le informazioni (che riesce ad ottenere durante la visita del grafo) utili a determinare i nodi ponte e le direzioni d'instradamento da inserire nelle due strutture. In ogni modo, anche non considerando le ottimizzazioni effettuate dall'algoritmo, assumendo di effettuare un numero di visite pari a una costante (vedi sottoparagrafo 6.2), il costo dell'individuazione dei nodi ponte e delle direzioni di instradamento è limitato da $\mathcal{O}(n \cdot m)$. Tale limite superiore è dato dal fatto che, al caso pessimo, per ogni arco e del grafo, si effettuano n confronti (uno per ogni nodo $v \in V$). I confronti servono a determinare quali nodi siano raggiungibili, tramite un cammino minimo, attraversando un dato arco e (il confronto è lo stesso effettuato per il controllo riportato nel sottoparagrafo 4.2.2). L'individuazione dei nodi raggiungibili con costo minimo per un dato arco, grazie alle strutture ausiliarie, tabella di routing e struttura dei *Jetsons*, porta alla determinazione dei ponti raggiungibili, attraversando un dato arco (vedi sottoparagrafo 5.1.3).

Individuate le associazioni tra archi e ponti e tra nodi destinazione e relativi ponti (associazione ottenuta usando la struttura dei *Jetsons*, vedi sottoparagrafo 5.1.3), il costo di costruzione dei filtri di Bloom e della tabella di routing equivale al numero di inserimenti effettuati nelle due strutture di dati. Per questo motivo, rimandiamo il calcolo del costo di inserimento al sottoparagrafo 6.2, che stima il numero di elementi presenti nei filtri di Bloom e nella tabella di routing. Per completezza nell'analisi del costo computazionale della fase di preprocessing, anticipiamo che il costo di costruzione dei filtri di Bloom è pari a $\mathcal{O}(m \cdot n \cdot \bar{p})$, dove $\bar{p}$ equivale al numero medio di ponti per ogni arco del grafo, mentre il costo di creazione della tabella è pari a $\mathcal{O}(n^2 \cdot \hat{p})$, dove $\hat{p}$ corrisponde al numero medio di ponti per ogni coppia di nodi (s, t), del grafo.

Dalle analisi sopra riportate, dopo aver osservato che $\hat{p}$ è una quantità trascurabile rispetto a n e a m[1], possiamo concludere che il costo computa-

[1]Fissata una coppia di nodi, sorgente-destinazione, il numero di ponti, corrispondenti

zionale della fase di preprocessing è pari a $\mathcal{O}(m \cdot n \cdot \bar{p})$[2], dove $\bar{p}$ equivale al numero medio di ponti per ogni arco del grafo.

6.2 Strutture di compressione

Come precedentemente illustrato (vedi sottoparagrafo 5.1.3) le strutture di compressione dei cammini minimi, costruite in fase di preprocessing, sono i DAG, la tabella di routing e i filtri di Bloom.

Il costo di occupazione della memoria da parte dei DAG e il costo computazionale delle visite in ampiezza effettuate per la loro creazione dipende dal numero di visite annidate del grafo (vedi sottoparagrafo 5.1.3). Stimare, attraverso un'analisi teorica, tale quantità è un problema non banale. Per questo motivo, tale stima è stata effettuata tramite un'analisi empirica.

Nella tabella sottostante sono riportati i risultati ottenuti, effettuando i passi ricorsivi di costruzione dei DAG annidati, al fine di stimare il numero medio di annidamenti.

Nome dataset	n	m	d	$nDAG$	$\frac{nDAG}{n}$	max_a	avg_a[3]
facebook_comb	4039	88234	8	14796	3.6	6	2
wiki-vote	7066	103689	7	11596	1.6	2	1
CA-HelpPh	11204	<118521[4]	13	59059	5.3	3	2
Kron14	8156	24506	4	11786	1.4	2	1.25
Forest1e4_2	10000	153925	3	28249	2.8	3	2.3
Darwin_Book	7725	46281	5	17050	2.2	2	1.2

Tabella 6.1: Tabella delle analisi sperimentali riguardanti l'occupazione in memoria dei DAG.

Come si può osservare (dalla tabella 6.1), il numero di DAG annidati, corrispondente al numero di ricorsioni (per ogni livello del grafo), tende ad essere "basso". Questo fa sì che la dimensione dei DAG rimanga nell'ordine

a tale coppia, è sempre minore o uguale al grado del nodo sorgente

[2]Calcolare il numero di ponti esistenti nel grafo e la loro posizione, in relazione a un dato nodo, è un problema non banale ed esula dagli scopi della nostra ricerca

[3]I parametri d, $nDAG$, $\frac{nDAG}{n}$, max_a e avg_a corrispondono, rispettivamente, ai valori del diametro, del numero di nodi dei DAG, dell'indice di crescita dell'occupazione di memoria (rispetto alla sola memorizzazione del grafo), al numero massimo di DAG annidati e al numero medio degli stessi.

[4]Il grafo non è completamente connesso, per il test si considera la sola componente connessa, formata da 11204 nodi. In totale i vertici del grafo sono 12008, mentre gli archi sono 118521

della dimensione del grafo stesso $\mathcal{O}(n+m)$ (si ricorda che ogni arco del grafo è mantenuto in uno ed in un solo DAG). Il basso numero di ricorsioni ci permette di considerare il numero di visite pari ad una costante; da questo derivano le assunzioni fatte al sottoparagrafo precedente. In realtà i DAG, costruiti i filtri di Bloom (nella fase di preprocessing), perdono la propria utilità, ma vengono comunque considerati nella descrizione della soluzione, per rendere più chiara la trattazione.

Analizzando il costo di costruzione dei filtri di Bloom, la prima osservazione da fare è che detto costo dipende dal numero di nodi creati nel DAG. Infatti, esiste un Bloom filter per ogni nodo di ogni DAG. Per questo motivo, le analisi empiriche fatte per i nodi dei DAG sono applicabili, in tutto e per tutto, anche ai filtri di Bloom. Dobbiamo, però, considerare che ogni Bloom filter rappresenta un insieme, anche se, si ricorda che, un Bloom filter occupa $M = -\frac{N \ln p}{(\ln 2)^2}$ *bit*, dove N è il numero di elementi dell'insieme e p è la probabilità di avere un falso positivo interrogando il filtro (vedi 2.4.3).

Per l'analisi di costo, in termini di tempo di costruzione e di spazio occupato, possiamo ignorare la struttura ad albero binario dei filtri; si veda [3] (in particolare il paragrafo riguardante l'unione insiemistica effettuata tramite filtri di Bloom). Consideriamo tutte, e sole, le foglie (si veda la struttura gerarchica descritta al paragrafo 5.1.3) dei filtri di Bloom creati per un dato nodo v del grafo G. Per ogni nodo v abbiamo tanti filtri di Bloom, quanti sono gli archi di v in G. Questi filtri contengono tutti i ponti necessari per raggiungere tutti i vertici del grafo G, a partire da v. Per questo motivo, il costo di creazione di tutti i Bloom Filter, per tutti i nodi v del grafo, equivale al numero di ponti memorizzati: $\mathcal{O}(m \cdot n \cdot \bar{p})$, dove $\bar{p}$ equivale al numero medio di ponti per ogni arco del grafo. Valutare la dimensione di $\bar{p}$ esula dagli scopi di questa tesi; tale valutazione potrebbe far parte di analisi successive, il cui scopo sia quello di ottimizzare la soluzione da noi individuata. Poiché, come precedentemente detto, ogni Bloom filter occupa un numero di bit proporzionale al numero di elementi dell'insieme che rappresenta, supposta l'esistenza di $\mathcal{O}(n)$ filtri (vedi risultati in tabella 6.1), in media ogni filtro di Bloom occupa $M = \mathcal{O}(m \cdot \bar{p} \cdot \ln p)$ *bit*. In totale i filtri occupano $\mathcal{O}(m \cdot n \cdot \bar{p} \cdot \ln p)$ *bit*.

La tabella di routing è un dizionario che, data una coppia di nodi (s, t) (sorgente-destinazione), restituisce i ponti da attraversare per recuperare tutti cammini minimi da s a t, a partire dai i nodi dei DAG[5], corrispondenti al nodo s del grafo. Per questo motivo, il dizionario contiene $\binom{n}{2} = \mathcal{O}(n^2)$ chiavi, con altrettanti valori. Ogni valore, corrispondente ad una generica

[5]Da notare che non è detto che esista un cammino minimo per t a partire da un qualsiasi nodo DAG, corrispondente al nodo s del grafo

chiave (s,t)[6], è costituito da una lista di coppie (s,l), dove s è un nodo di un DAG, corrispondente al nodo $s \in G$, ed l è la lista dei ponti usati a partire da s (nodo del DAG) fino a raggiungere t (nodo di un DAG, corrispondente al nodo t del grafo). La cardinalità dell'insieme formato dagli elementi delle liste l, corrispondenti alla chiave (s,l), è limitata dal grado del nodo $s \in G$. Da notare però che, per costruzione, si cerca di limitare ad uno il numero di ponti presenti in ogni lista l della tabella (vedi sottoparagrafo 5.1.3, figura 5.2), anche se in alcuni casi non è possibile: se in un DAG più archi di s portano a più ponti per uno stesso nodo t, è necessario inserire tutti i ponti all'interno della lista l (vedi figura 5.2).

I risultati empirici, riguardo il numero di annidamenti dei DAG, forniscono, anche in questo caso, delle informazioni utili all'analisi dei costi. Poiché possiamo considerare trascurabile (rispetto a n e m) il numero di ricorsioni, il numero di coppie (s,l) ,per ogni chiave (s,t), diventa anch'esso irrilevante. Rimane da quantificare il numero di elementi presenti in l per ogni chiave (s,t). Stimare tale valore non è banale, perché dipende strettamente dalle connessioni del grafo. Semplificando l'analisi, poiché attuiamo una strategia che cerca di limitare ad uno la lunghezza delle liste l, possiamo valutare la dimensione della tabella di routing nell'ordine di $\mathcal{O}(n^2)$, numero di chiavi del dizionario.

Concludendo, possiamo affermare che le strutture di dati usate per la memorizzazione implicita di tutti i cammini minimi constano del grafo che occupa $\mathcal{O}(m+n)$, dei filtri di Bloom occupanti $\mathcal{O}(m \cdot n \cdot \bar{p} \cdot \ln p)$ *bit* e della tabella di routing, memorizzata con un costo pari a $\mathcal{O}(n^2)$.

[6]In questo caso s e t sono due identificatori univoci corrispondenti a due nodi del grafo.

Capitolo 7

Conclusioni

Lo studio alla base della ricerca, le proprietà di ridondanza individuate[1] e l'analisi di diverse possibili soluzioni (vedi capitolo 4) ha portato allo sviluppo di una soluzione output sensitive, per il problema di recupero di tutti i cammini minimi esistenti per una coppia di nodi (s, t). Il processo di recupero dei cammini minimi (vedi sottoparagrafo 5.1.4) è costituito da più funzioni ricorsive (vedi algoritmi 12 e 13, sezione 5.3), il cui costo dipende dal numero di cammini minimi esistenti da s a t. L'unico costo aggiuntivo è dato dai possibili cammini "imboccati" erroneamente, a causa dei falsi positivi generati dai filtri di Bloom, ma tale costo può essere controllato dal parametro p, aumentando o riducendo la dimensione dei filtri (vedi sottosezione 2.4.3). In ogni modo, anche se, al fine di comprimere lo spazio usato dai filtri decidiamo di aumentare p, la probabilità di avere di nuovo un falso positivo al passo successivo è ancora p (le probabilità dei filtri di Bloom sono tra loro disgiunte). La probabilità di proseguire erroneamente il cammino diminuisce esponenzialmente con il numero di passi effettuati.

7.1 Sviluppi futuri

Lo studio effettuato non esaurisce l'argomento trattato, ma, piuttosto, apre nuove strade interessanti di ricerca. Di seguito riportiamo quelle da noi individuate:

Estensione ai grafi pesati
: È interessante valutare la possibile estensione della soluzione ai grafi pesati (sostituendo le visite BFS con visite tramite l'applicazione di Dijkstra).

[1] Si veda il concetto di ridondanza applicato all'insieme dei cammini minimi, descritto nel capitolo 3

Variante 1: unione di più livelli
Valutare i vantaggi/svantaggi che si possono avere effettuando i passi ricorsivi su nodi a distanze diverse (fondendo più livelli[2] vicini).

Variante 2: analisi dell'attuale strategia di routing e modifica della stessa

Analizzare teoricamente o tramite uno studio empirico la dimensione della tabella di routing. Studiare una strategia di instradamento diversa. La tabella di routing contiene informazioni ulteriormente comprimibili; infatti, può succedere che uno stesso nodo sia un nodo ponte per una data destinazione, per più nodi sorgente. Nell'attuale soluzione, l'informazione, riguardante il ponte, è memorizzata più volte in tabella.

Variante 3: comprimere i filtri di Bloom
Studiare la possibilità di unire "virtualmente" più DAG, unendo i filtri di Bloom relativi alle diverse copie (presenti in più DAG) di uno stesso nodo del grafo. Durante l'unione dei DAG, è importante evitare di costruire degli indirizzamenti ciclici: da un ponte b_1 al ponte b_2, per poi tornare in b_1.

Variante 4: minimizzare il falsi positivi dei filtri di Bloom
Studiare una modalità per unire i filtri di Bloom in strutture gerarchiche ad albero, minimizzando la probabilità di errore, attraverso una strategia simile a quella usata in teoria dell'informazione per generare il codice di Huffman.

Analisi sperimentali
Valutare, in modo più approfondito, la capacità della soluzione sviluppata, di comprimere i cammini minimi su grafi reali.

Analisi teoriche
Studiare la distribuzione (all'interno del grafo) dei nodi ponte potrebbe rivelarsi importante per individuare proprietà di connessione. I nodi ponte svolgono un compito simile a quello svolto dai *landmark* [5].

[2]Ogni livello del DAG è formato dai nodi equidistanti dal nodo (radice) da cui inizia la visita BFS e dai relativi archi uscenti

Glossario

APSP All Pairs Shortest Path
ricercare un cammino minimo per tutte le coppia di nodi di un grafo

BDD Binary Decision Diagram
Diagramma binario di decisione

BFS Breadth-First Search
Ricerca di un nodo, a partire da una sorgente ben definita, visitando il grafo in ampiezza

DAG Directed Acyclic Graph
grafo diretto privo di cicli

FIFO First In First Out
modalità di immagazzinare elementi, in una coda, in modo tale che il primo elemento introdotto sia il primo ad uscire

MSSP Multiple Source Shortest Path
ricercare un cammino minimo per ogni nodo del grafo, dato un insieme di nodi sorgente

SSSP Single Source Shortest Path
ricercare un cammino minimo per ogni nodo del grafo, dato un nodo sorgente

Bibliografia

[1] G. Bigi, A. Frangioni, G. Gallo, S. Pallottino, and M. G. Scutellà. *Appunti di Ricerca Operativa*. SEU Servizio Editoriale Universitario di Pisa, 2006.

[2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 385–509, 2004.

[4] P. Crescenzi, G. Gambosi, and R. Grossi. *Strutture di dati e algoritmi: progettazione, analisi e visualizzazione*. Pearson Education Italia, 2006.

[5] Daniel Delling and Dorothea Wagner. Landmark-based routing in dynamic graphs. In *IN: 6TH WORKSHOP ON EXPERIMENTAL ALGORITHMS*. Springer, 2007.

[6] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, pages 968–992, 2004.

[7] Changxing Dong and Paul Molitor. What graphs can be efficiently represented by bdds? In *ICCTA*, pages 128–134. IEEE Computer Society, 2007.

[8] David Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.

[9] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006.

[10] Jittat Fakcharoenphol and Satish Rao. Shortest paths in planar graphs with negative weight edges. In *Encyclopedia of Algorithms*. 2008.

[11] Paolo Ferragina, Igor Nitto, and Rossano Venturini. On compact representations of all-pairs-shortest-path-distance matrices. *Theoretical Computer Science (TCS)*, 411(34-36):3293–3300, 2010.

[12] B. L. Fox. More on kth shortest paths. 1975.

[13] Zvi Gotthilf and Moshe Lewenstein. Improved algorithms for the k simple shortest paths and the replacement paths problems. *Inf. Process. Lett.*, 2009.

[14] Jun Kawahara Hiroaki Iwashita and Shin ichi Minato. Zdd-based computation of the number of paths in a graph. Technical report, Hokkaido University Graduate School of Information Science and Technology, 2012.

[15] Jun Kawahara Takeaki Uno Hiroaki Iwashita, Yoshio Nakazawa and Shin ichi Minato. Efficient computation of the number of paths in a grid graph with minimal perfect hash functions. Technical report, Hokkaido University Graduate School of Information Science and Technology, 2013.

[16] J.W. Hui, J.P. Vasseur, and W. Hong. Compressing data packet routing information using bloom filters, February 6 2014. US Patent App. 13/563,077.

[17] Philip N. Klein, Shay Mozes, and Christian Sommer. Structured recursive separator decompositions for planar graphs in linear time. *CoRR*, abs/1208.2223, 2012.

[18] Philip N. Klein, Shay Mozes, and Oren Weimann. Shortest paths in directed planar graphs with negative lengths: a linear-space $o(n \log^2 n)$-time algorithm. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 236–245, 2009.

[19] Donald Knuth. Simpath. `http://www-cs-faculty.stanford.edu/~uno/programs/simpath.w`. Accessed: 2014-04-07.

[20] Donald E. Knuth. *The Art of Computer Programming, Volume 4A, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2011.

[21] E.L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 1972.

[22] Shay Mozes and Christian Sommer. Exact distance oracles for planar graphs. In *SODA*, pages 209–222, 2012.

[23] Toshihide Ibaraki Naoki Katoh and Hisashi Mine. An efficient algorithm for K shortest simple paths. *Networks*, 12(4):411–427, 1982.

[24] Liam Roditty and Uri Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. *ACM Transactions on Algorithms*, 8(4):33, 2012.

[25] Christian Sommer. *Approximate Shortest Path and Distance Queries in Networks*. PhD thesis, The University of Tokyo, 2010.

[26] Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46, 2014.

[27] Jun Kawahara Takeru Inoue, Hiroaki Iwashita and Shin ichi Minato. Graphillion: Software library designed for very large sets of graphs in python. Technical report, Hokkaido University Graduate School of Information Science and Technology, 2013.

[28] Shogo Takeuchi, Jun Kawahara, Akihiro Kishimoto, and Shin ichi Minato. Shared-memory parallel frontier-based search. In *WALCOM*, pages 170–181, 2013.

[29] J.Y. Yen. Finding the k shortest loopless paths in a network. *management Science*, 1971.

[30] Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by zdds. *Algorithms*, 5:176–213, 2012.

Ringraziamenti

La sezione dei ringraziamenti è veramente la parte della tesi più difficile da scrivere. Questi anni di università portano con sé tanti ricordi e soprattutto tante persone.

Prima di tutto ringrazio la mia famiglia che c'è sempre, mi supporta e mi sopporta, senza la quale non avrei avuto la possibilità di studiare.

Un grazie ai professori universitari, che mi hanno istruito all'"arte dell'informatica", in particolare al professor Grossi.

Poi tutti i miei amici, quelli che lo sono da una vita Anna, Ele, Federica, Martina, Sara, Elisa, Elsa, Luisa, Veronica, Andrea, Edo, Matteo, Marco, Rudy, Samuele.

Grazie a tutti gli amici di cormorano ed emporio: Marta (ringrazio la trigonometria che ci ha fatto incontrare), Rita (ed Aibo), Ale, Fabio, Fede, Marzio, Michael (anche per il lavoro su carpe diem), Gaggo (per il supporto da remoto) e Roby. Tutti gli amici di questi anni in università, perché continuano ad esserci nonostante le distanze e i diversi impegni: Nicola, Stefano, Laura, Fabio, Marco e Fabio (per avermi ascoltata ed aiutata durante i miei ragionamenti sulla lavagna magnetica), Catia, Davide, Hind e Emilio. Tutti coloro con cui ho dato un esame o non l'ho dato, ma con cui ho comunque condiviso la fatica dello studio.

Ringrazio anche la famiglia scout: tutta la comunità capi nuova/vecchia e anche futura ed i ragazzi (anche se probabili non lo sapranno mai).
Grazie a Dania e Francesca e il gruppo giovani tutto.
Un grazie a tutti i "compagni di viaggio" e "di strada".