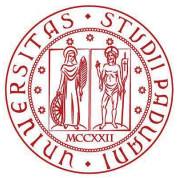

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

Corso di Laurea Triennale in Matematica

Riottimizzare cammini minimi:

analisi e strategie risolutive

Relatrice:
Prof.ssa Carla De Francesco

Laureando: Martino Zaratin
Matricola: 2006267

Anno Accademico 2022/2023

22 Settembre 2023

Indice

Introduzione

In letteratura scientifica, nell'ambito dell'ottimizzazione combinatoria, il *problema del cammino minimo* è una tematica a cui si è sempre dedicata grande attenzione. Dato un grafo orientato in cui a ciascun arco è stato assegnato un costo, l'obiettivo di un classico problema di cammino minimo consiste nel determinare un cammino di costo minimo avente come estremi due nodi scelti all'interno del grafo. Una naturale estensione di tale quesito è il *problema dell'albero dei cammini minimi*, in cui, a partire da un nodo detto radice, si cercano i cammini minimi aventi come destinazione tutti i restanti nodi del grafo.
I motivi per cui questi problemi vengono trattati così frequentemente sono diversi. In primo luogo la loro importanza si nota da un punto di vista teorico. Infatti per risolvere problemi di natura combinatoria spesso si passa per il calcolo di cammini minimi oppure si fa uso di concetti che appartengono al contesto dell'ottimizzazione del costo di cammini su grafi orientati.
Un secondo motivo che rende questi problemi particolarmente rilevanti è il loro utilizzo negli ambiti applicativi. Ci riferiamo a tutti quei contesti in cui si trattano reti di grandi dimensioni (*large-scale networks*) e si rende necessaria l'ottimizzazione di certi percorsi su queste strutture. Basti pensare, ad esempio, all'analisi e la programmazione di trasporti su larga scala, problemi di gestione del traffico urbano o la dinamica dell'instradamento di veicoli in determinate reti stradali. Il problema del cammino minimo compare anche nella logistica della comunicazione, così come può essere utile per redigere piani di emergenza in caso di calamità improvvise, come spiegato in [10].
Negli anni sono stati sviluppati algoritmi risolutivi di notevole efficienza: primo fra tutti fu quello ideato da E. Dijkstra nel 1956, che calcola l'albero dei cammini minimi nel caso in cui i costi degli archi siano non negativi. Successivamente significativi progressi sono stati ottenuti grazie alla collaborazione tra ricercatori di Ricerca Operativa e Computer Science. Questi ultimi hanno messo a disposizione tecniche e strutture dati sofisticate che hanno portato a miglioramenti considerevoli nell'implementazione degli algoritmi per il calcolo degli alberi di cammini minimi.

In diversi contesti applicativi si rende necessario risolvere non uno, bensì una sequenza di problemi dell'albero dei cammini minimi. Si pensi ad esempio al caso di una rete stradale in cui la configurazione del traffico si evolve nel tempo e in ciascun istante è richiesto di trovare il percorso più veloce tra due punti della mappa. In queste situazioni, due istanze successive del problema solitamente differiscono per un piccolo cambiamento nella struttura del grafo: ad esempio può avvenire che alcuni archi aumentino o diminuiscano di costo, può cambiare la radice oppure possono venire inseriti o rimossi alcuni archi.

Due sono i possibili approcci alla risoluzione di questo tipo di problemi: il primo consiste nell'utilizzare ripetutamente gli algoritmi per il cammino minimo, risolvendo ciascuna istanza del problema ex novo; la seconda strategia è fare uso delle tecniche di *riottimizzazione*. L'obiettivo della riottimizzazione consiste nel risolvere il k-esimo problema della sequenza sfruttando le informazioni ottenute dalla soluzione del $(k-1)$-esimo quesito. Nonostante la notevole efficienza degli algoritmo risolutivi per la ricerca di cammini minimi come l'algoritmo di Dijkstra o quello di Bellman-Ford, mai è diminuito l'interesse nei confronti delle tecniche di riottimizzazione. Per ogni specifico contesto sono nati algoritmi che si sono resi competitivi rispetto alle procedure classiche di risoluzione ex novo. Ad esempio, G. Gallo in [5] propose un primo algoritmo di riottimizzazione efficiente in cui viene affrontato il seguente problema: supponiamo di conoscere un albero dei cammini minimi, sia esso T_r^*, soluzione ottima rispetto ad una radice r. Vogliamo calcolare la nuova soluzione ottima dopo che o avviene un cambio della radice oppure il costo di esattamente un arco diminuisce (in realtà, il secondo problema costituisce una naturale generalizzazione del primo). Un altro interessante approccio fu quello studiato da Fujishige (si veda [4]) per il caso in cui viene diminuito il costo di tutti gli archi incidenti in un nodo comune.

In questa tesi si fa riferimento al contributo di Stefano Pallottino e Maria Grazia Scutellà i quali in [8] formulano un algoritmo risolutivo per il problema di riottimizzazione nel contesto più generale possibile, ovvero quello in cui un qualunque sottoinsieme di archi del grafo di partenza è soggetto ad una variazione di costo, che può verificarsi come un aumento, una diminuzione o entrambi. Dopo un'introduzione ai concetti di base necessari per costruire questa teoria, formuleremo il problema nel linguaggio della programmazione lineare e indagheremo il funzionamento di alcuni algoritmi risolutivi in più casi, dai più semplici ai più complessi, per arrivare a definire una strategia generale. Tutto ciò verrà corredato da esempi in cui verranno messi in pratica i concetti espressi nella parte teorica. Dimostreremo la correttezza delle procedure implementate e spiegheremo perché, almeno da un punto di vista teorico, questa strategia si dimostra competitiva rispetto alla classica risoluzione ex novo.

Capitolo 1

Definizioni e risultati di base

Per tutta la lunghezza della discussione verranno utilizzati concetti di teoria dei grafi e di programmazione lineare. In questa sezione, dunque, richiamiamo alcune nozioni fondamentali riguardanti i seguenti argomenti: prima alcune definizioni e notazioni di teoria dei grafi, poi alcuni risultati di base sulla teoria della dualità per la programmazione lineare, infine alcune importanti considerazioni sulle matrici unimodulari, che risulteranno necessarie in seguito.

1.1 Teoria dei grafi

Un **grafo (non orientato)** G è una coppia ordinata $G = (V, E)$ dove V è un insieme finito non vuoto ed E è un insieme di coppie non ordinate di elementi distinti di V, cioè $E \subseteq \{(i, j) \colon i, j \in V, i \neq j\}$. Chiamiamo gli elementi di V vertici (o nodi); chiamiamo gli elementi di E spigoli.
Dato un grafo $G = (V, E)$, consideriamo una sequenza alternata di nodi e spigoli di G: $v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k$, con $k \geq 0$. Se ciascun arco ha come estremi due nodi consecutivi della sequenza, cioè $e_i = (v_{i-1}, v_i) \in E$ per ogni $i = 1, \ldots, k$ e tutti i nodi sono distinti, allora chiamiamo tale sequenza un **cammino**. Se invece vale che tutti i nodi sono distinti tranne il primo e l'ultimo, cioè $v_0 = v_k$, allora la sequenza prende il nome di **ciclo**.
Un grafo si dice **connesso** se per ogni coppia di nodi esiste un cammino avente tali nodi come estremi.
Un **albero** è un grafo connesso e aciclico, cioè privo di cicli.
Sia $G = (V, E)$ un grafo non orientato. Un albero $T \subseteq G$ si dice **ricoprente** se contiene tutti i nodi del grafo; tale albero ha la proprietà che, per ogni coppia di nodi $(i, j) \in N$ esiste (ed è unico) un cammino contenuto in T avente come estremi i e j.

Un **grafo orientato** D è una coppia ordinata $D = (V, A)$ di insiemi finiti V ed A, dove V è detto l'insieme dei nodi e $A \subseteq \{(i, j) \in V \times V \colon i \neq j\}$ è detto l'insieme degli archi di D. In un grafo orientato gli archi sono coppie *ordinate* di nodi e hanno una rappresentazione univoca: in particolare $(i, j) \neq (j, i)$.
Ad un grafo orientato soggiace sempre un grafo non orientato, il cosiddetto **grafo sottostante**: si ottiene sostituendo agli archi i corrispondenti spigoli e ignorando eventuali ripetizioni ottenute. Risulteranno fondamentali le seguenti definizioni: dato un nodo

$i \in V$, denotiamo con $FS(i)$ (*Forward Star*) l'insieme degli archi uscenti da i e con $BS(i)$ (*Backward Star*) l'insieme degli archi entranti in i:

$$FS(i) = \{(u, v) \in A \colon u = i\} \qquad BS(i) = \{(u, v) \in A \colon v = i\}$$

Un **percorso orientato** è una sequenza di nodi e archi $v_0, e_1, v_1, \ldots, v_n$, $n \geq 1$, dove $e_k = (v_{k-1}, v_k) \in A$ per ogni $k = 1, \ldots, n$. Un percorso orientato è detto un **cammino orientato** se nessun nodo si ripete nella sequenza. Infine, se $k \geq 2$ e l'unica ripetizione si ha per $v_0 = v_n$, in tal caso si parla di **ciclo orientato**.
Un grafo orientato si dice **fortemente connesso** se esiste un cammino orientato tra ciascuna coppia di nodi.
Un **albero** (orientato) è un grafo a cui soggiace un albero. Nel seguito, siccome tratteremo unicamente grafi orientati, chiameremo semplicemente "alberi" questi grafi intendendoli orientati.
Infine, dato un grafo orientato $D = (V, A)$ e un nodo $r \in V$, D è un'**arborescenza** con radice in r se $|A| = |V| - 1$ e per ogni $v \in V$ esiste un cammino orientato da r a v. Si noti che ogni arborescenza è un albero e che, in un'arborescenza D con radice r, il cammino orientato in D che collega r ad ogni altro nodo di D è unico.

1.2 Teoria della dualità

In programmazione lineare ricopre un ruolo fondamentale la teoria della dualità. L'idea che sta alla base di questa teoria consiste nell'associare ad ogni problema di programmazione lineare un altro problema opportunamente definito, chiamato *duale*; per contro, il problema originario viene detto *primale*. Dato un programma lineare P, il corrispondente duale, sia esso D, avrà tante variabili quanti i vincoli di P e tanti vincoli quante le variabili di P. Esistono delle regole ben precise per la costruzione di D: per maggiori dettagli si faccia riferimento a [1].
Diverse sono le interpretazioni che si possono dare ai programmi duali, le quali variano a seconda del contesto in cui ci troviamo. In generale però l'utilità del problema duale risiede nel fatto che le soluzioni primali e quelle duali sono strettamente legate (si pensi ad esempio al Teorema di Dualità Debole o al Teorema di Dualità Forte, come descritto in [1]). Ad esempio, dati P e D due programmi lineari, dove D è il duale di P, e data x^* soluzione ammissibile per P, una soluzione ottima y^* di D rappresenta un certificato di ottimalità per x^*, nel senso che, verificando semplici uguaglianze, possiamo stabilire se la soluzione primale è ottima o non lo è. Un importante risultato che dà utili informazioni riguardo la relazione tra un programma lineare e il suo duale è il seguente teorema.

Teorema 1.2.1 (Teorema degli scarti complementari)**.** *Si consideri un programma lineare generico: sia $\bar{x}$ una sua soluzione ammissibile e sia $\bar{y}$ una soluzione ammissibile per il suo duale. $\bar{x}$ e $\bar{y}$ sono ottime per i rispettivi problemi se e solo se valgono le condizioni degli scarti complementari:*

- *Per ogni coppia vincolo primale-variabile duale, $\bar{x}$ soddisfa il vincolo a uguaglianza oppure la corrispondente variabile duale è uguale a 0.*

- *Per ogni coppia variabile primale-vincolo duale, $\bar{y}$ soddisfa il vincolo a uguaglianza oppure la corrispondente variabile primale è uguale a 0.*

1.3 Programmazione lineare intera e matrici unimodulari

Consideriamo ora un gruppo di problemi specifico: i programmi lineari interi. Un programma lineare si dice **intero** se la regione ammissibile è costituita dai vettori che, oltre a soddisfare un certo sistema di equazioni e/o disequazioni lineari, soddisfano anche il vincolo aggiuntivo di interezza delle componenti. Se rappresentiamo il sistema lineare sopra citato nella forma compatta $Ax \sim b$, dove $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$ e "$\sim$" indica, per ciascuna riga, uno qualunque degli operatori "$\geq$", "$\leq$" o "$=$", allora un programma lineare intero (di massimo) si presenta così:

$$\begin{aligned} \max \quad & c^\top x \\ s.a \quad & Ax \sim b \\ & x \in \mathbb{Z}^n \end{aligned}$$

dove a "max" sostituiamo "min" se il problema è di minimo.
In generale, i problemi di programmazione lineare intera risultano più complicati rispetto ai problemi di programmazione lineare. Esistono diversi approcci risolutivi che hanno portato allo sviluppo di algoritmi più o meno performanti a seconda del problema che si presenta di volta in volta.
Dato un programma lineare intero, può essere ad esempio utile considerare il corrispondente **rilassamento lineare**: si tratta del medesimo programma lineare in cui si ignorano i vincoli di interezza sulla soluzione. Essendo questo un programma lineare a tutti gli effetti, si può risolvere con i metodi classici, come ad esempio il metodo del simplesso. Tuttavia nulla assicura che la soluzione trovata per il rilassamento lineare di un problema sia una buona approssimazione della soluzione del problema originale. Si possono addirittura costruire esempi di programmi lineari interi in cui la soluzione ottima è arbitrariamente distante dalla soluzione del corrispondente rilassamento lineare. È dunque necessario abbandonare completamente questa strategia? In realtà, sotto opportune ipotesi sulla matrice dei vincoli, si può dimostrare che il rilassamento lineare porta ad una corretta risoluzione del problema di programmazione intera.

Definizione 1.3.1. Una matrice $A \in \mathbb{Z}^{m\times n}$, con $m \leq n$ si dice **unimodulare** se ogni sottomatrice $m \times m$ di A ha determinante 0, 1 o -1.

Proposizione 1.3.1. *Sia $A \in \mathbb{Z}^{m\times n}$ una matrice con entrate intere. Le soluzioni ammissibili di base del sistema (in forma standard)*

$$\begin{aligned} Ax &= b \\ x &\geq 0 \end{aligned}$$

sono tutte intere qualsiasi sia $b \in \mathbb{Z}^m$ se e solo se A è unimodulare.

Dimostrazione. Sia A una matrice unimodulare. Consideriamo una qualche base $B \subseteq \{1, \ldots, n\}$ e la relativa soluzione di base $\bar{x}$: essa è della forma $\bar{x}_B = A_B^{-1}b$, $\bar{x}_N = 0$. A_B è invertibile e, per ipotesi di unimodularità di A, $|\det A_B| = 1$. Utilizzando ad esempio il metodo dei complementi algebrici per calcolare la matrice inversa, si deduce che A_B^{-1} ha tutte le entrate intere. Questo fatto, unito alla scelta di b intero, comporta che anche il vettore $\bar{x}$ è intero, qualunque sia $b \in \mathbb{Z}^m$.

Viceversa, supponiamo che le soluzioni ammissibili di base del sistema siano intere per ogni scelta del vettore $b \in \mathbb{Z}^m$ e dimostriamo che A è unimodulare.
Per farlo dimostriamo prima un risultato intermedio: con tali ipotesi vale che ogni sottomatrice $m \times m$ di A non singolare ha inversa intera. Sia B una sottomatrice $m \times m$ di A e sia $B^{-1} = [C_1, \ldots, C_m]$, dove C_k indica la k-esima colonna di B^{-1}. Dimostriamo che una generica colonna k-esima C_k è intera.
Sia t un vettore intero tale che $t + C_k \geq 0$ e definiamo $b(t) = Bt + e_k$, dove e_k rappresenta il k-esimo vettore unitario. Osserviamo che

$$B^{-1}b(t) = B^{-1}(Bt + e_k) = t + B^{-1}e_k = t + C_k \geq 0$$

Ciò significa che $t + C_k$ è soluzione di base ammissibile del sistema $Ax = b(t)$. Per ipotesi allora $t + C_k$ è intero e dunque, avendo scelto t intero, anche C_k lo è. Siccome la scelta dell'indice k non è stata in alcun modo restrittiva, possiamo concludere che tutte le colonne di B^{-1} sono intere, cioè B^{-1} è intera.

Ora osserviamo che, per quanto appena dimostrato, $|\det B|$ e $|\det B^{-1}|$ sono interi. Ma siccome

$$|\det B||\det B^{-1}| = |\det(BB^{-1})| = 1$$

necessariamente $|\det B| = |\det B^{-1}| = 1$. Ciò permette di concludere che A è unimodulare. □

Questo risultato ha la seguente importante conseguenza: dato un programma lineare intero in forma standard

$$\begin{array}{ll} \max & c^\top x \\ \text{s. a} & Ax = b \\ & x \geq 0 \\ & x \in \mathbb{Z}^n \end{array} \tag{1.1}$$

se A è unimodulare e b è intero, allora ogni soluzione ottima del suo rilassamento lineare, se esiste, è intera. Ciò implica che applicando il metodo del simplesso al rilassamento lineare del problema si giunge effettivamente alla soluzione cercata.

Richiedendo condizioni più stringenti si può generalizzare la discussione appena fatta al caso di programmi lineari non in forma standard, dove i vincoli sono disequazioni con "$\leq$" e le variabili sono non negative. In particolare è necessario (e sufficiente) chiedere che la matrice A dei vincoli abbia la seguente proprietà:

Definizione 1.3.2. Una matrice si dice **totalmente unimodulare** se tutte le sue sottomatrici quadrate, di ogni ordine, hanno determinante uguale a 0, 1 o -1.

Per dimostrare il teorema poco fa anticipato, è necessario il seguente lemma.

Lemma 1.3.1. *Una matrice $A \in \mathbb{Z}^{m\times n}$ è totalmente unimodulare se e solo se $[A|\mathbb{I}_m]$ è unimodulare, con $\mathbb{I}_m$ matrice identica $m \times m$.*

Teorema 1.3.1 (Hoffman-Kruskal)**.** *Sia $A \in \mathbb{Z}^{m\times n}$. Le soluzioni ammissibili di base del sistema*

$$\begin{aligned} Ax &\leq b \\ x &\geq 0 \end{aligned} \tag{1.2}$$

sono tutte intere qualunque sia $b \in \mathbb{Z}^m$ se e solo se A è totalmente unimodulare.

Dimostrazione. Definito un vettore di variabili di scarto non negative s, poniamo il sistema 1.2 in forma standard: otteniamo

$$\begin{aligned} Ax + s &= b \\ x \geq 0, s &\geq 0 \end{aligned}$$

In base alla Proposizione 1.3.1, le soluzioni ammissibili di base di questo sistema sono tutte intere per ogni $b \in \mathbb{Z}^m$ se e solo se la matrice dei vincoli è unimodulare. La matrice dei vincoli è $[A|\mathbb{I}_m]$, la quale per il Lemma 1.3.1 è unimodulare se e solo se A è totalmente unimodulare. Da questa catena di doppie implicazioni segue la tesi. □

Grazie a questo teorema possiamo concludere che, dato un programma lineare intero non in forma standard del tipo

$$\begin{aligned} \max \quad & c^\top x \\ \text{s. a} \quad & Ax \leq b \\ & x \geq 0 \\ & x \in \mathbb{Z}^n \end{aligned}$$

con A totalmente unimodulare e b intero, le sue soluzioni sono intere. Ciò comporta inoltre che applicare il metodo del simplesso al rilassamento lineare del problema porta ad una corretta risoluzione del programma.

Concludiamo la sezione con una caratterizzazione delle matrici totalmente unimodulari. Data una matrice $A \in \{0, \pm 1\}^{m\times n}$, supponiamo di avere a disposizione due colori, rosso e blu, e di colorare ciascuna colonna di A con uno di questi due colori (formalmente si tratta di una partizione dell'insieme $\{1, \ldots, n\}$ in due sottoinsiemi). Una **bicolorazione equa delle colonne di A** è una colorazione per cui vale la seguente proprietà: per ogni riga di A, se la somma degli elementi della riga è pari, allora la somma degli elementi rossi eguaglia quella degli elementi blu; se la somma totale degli elementi della riga è invece dispari, la somma degli elementi rossi e quella degli elementi blu differiscono esattamente di 1.
Vale il seguente teorema:

Teorema 1.3.2 (Ghouila-Houri)**.** *Una matrice $A \in \{0, \pm 1\}^{m \times n}$ è totalmente unimodulare se e solo se ogni sottomatrice di A ammette una bicolorazione equa delle sue colonne.*

Osservazione 1. *Siccome una matrice è totalmente unimodulare se e solo se lo è la sua trasposta, si può definire un concetto simmetrico a quello sopra descritto che riguarda la bicolorazione equa delle righe, in cui i ruoli di righe e colonne sono invertiti. Chiaramente, il Teorema 1.3.2 vale nella sua versione simmetrica, formulata per la bicolorazione equa delle righe.*

Capitolo 2

Il problema dell'albero dei cammini minimi

Presentiamo ora il problema dell'albero dei cammini minimi (*Shortest Path Tree problem*) nella sua formulazione matematica. Dopo alcune considerazioni sulle condizioni di ottimalità delle soluzioni, seguirà la descrizione di un algoritmo risolutivo, l'algoritmo di Dijkstra. Infine verrà affrontata la questione riguardante la sua implementazione e le relative conseguenze sul costo computazionale.

2.1 Formulazione matematica

Consideriamo un grafo orientato $G = (N, A)$, dove $|N| = n$ e $|A| = m$. Sia $c\colon A \to \mathbb{R}$ una funzione che assegna ad ogni arco $(i, j) \in A$ un *costo* reale c_{ij}. Il costo di un sottoinsieme di archi $S \subseteq A$ è la quantità $c(S) = \sum_{(i,j)\in S} c_{ij}$. Facciamo le seguenti ipotesi sulla struttura di G:

1. G non contiene archi paralleli: per ogni coppia ordinata di nodi connessi da un arco esiste esattamente un arco che li collega;

2. G è fortemente connesso. Tale assunzione non è restrittiva poiché se non esiste un arco $(i, j) \in A$ per $i, j \in N$ possiamo comunque aggiungere tale arco ad A assegnandogli un costo $c_{ij} = M$ con $M \gg 0$ e da un punto di vista algoritmico il problema rimarrà invariato dopo la modifica.

Dato un nodo $r \in N$, il *problema dell'albero dei cammini minimi* con radice r consiste nel trovare un albero ricoprente T_r^* tale che, per ogni nodo $i \in N$, il cammino da r a i contenuto nell'albero è un cammino di costo minimo da r a i in G. Di fatto si può considerare come una generalizzazione del *problema del cammino minimo*, in quanto l'albero cercato è l'unione dei cammini di costo minimo da r a tutti i rimanenti nodi del grafo.

Come nel caso del problema del cammino minimo, è di fondamentale importanza assumere che il grafo G **non contenga alcun ciclo di costo negativo** e che ogniqualvolta si consideri un algoritmo che opera su un sottoinsieme degli archi o dei nodi, si sia certi

che questa proprietà sia mantenuta. Essa infatti garantisce la limitatezza delle soluzioni e dunque l'ottimalità e la terminazione dell'algoritmo stesso.

Per formulare il problema usando la notazione della programmazione lineare, assegniamo ad ogni arco (i, j) una variabile x_{ij} nel seguente modo: x_{ij} "conta" il numero di cammini minimi di cui l'arco (i, j) fa parte. Dunque

$$x_{ij} = |\{\text{P}\colon \text{P è un cammino minimo da } r \text{ a } v,\ \forall\, v \in G,\ (i,j) \in P\}|$$

Osserviamo che un albero di cammini minimi ha le seguenti proprietà:

1. dalla radice r escono esattamente $n-1$ cammini minimi, mentre non vi sono archi entranti;

2. ogni nodo diverso dalla radice è terminazione esattamente di un cammino minimo. Può succedere che un nodo sia un nodo intermedio di un cammino di cui non è terminazione e in tal caso vi sono tanti cammini uscenti quanti entranti, oltre al cammino di cui è terminazione.

In vista di queste considerazioni, possiamo scrivere il seguente programma lineare primale:

$$\begin{aligned} \min \quad & \sum_{(i,j)\in A} c_{ij}x_{ij} \\ \text{s. a} \quad & \sum_{(j,i)\in BS(i)} x_{ji} - \sum_{(i,j)\in FS(i)} x_{ij} = \begin{cases} -n+1, & i = r \\ 1, & i \in N \setminus \{r\} \end{cases} \\ & x_{ij} \geq 0 \quad \forall (i,j) \in A \\ & x_{ij} \in \mathbb{Z} \quad \forall (i,j) \in A \end{aligned} \tag{2.1}$$

Tale programma è scritto per m variabili, una per ogni arco del grafo.
Osserviamo ora la matrice dei vincoli. Si noti che ogni colonna ha esattamente due entrate non nulle, una occupata da un 1, una da un -1. Infatti la colonna della variabile x_{ij} avrà un -1 nella componente corrispondente alla riga del nodo i e un 1 nella componente relativa alla riga del nodo j. Questa proprietà è di fondamentale importanza perché implica che A ed ogni sua sottomatrice ammettono una bicolorazione equa delle righe: basta colorare tutte le righe dello stesso colore. Per il Teorema 1.3.2, si ha che la matrice A è totalmente unimodulare e dunque unimodulare. Essendo il vettore dei termini noti intero, ciò comporta che, per la Proposizione 1.3.1, l'ultimo vincolo riguardante l'interezza delle soluzioni è di fatto superfluo poiché il rilassamento lineare del programma restituisce esso stesso soluzioni intere.

Per formulare ora il programma duale, ci serviamo di n variabili $\pi_1, \ldots, \pi_n$, ciascuna corrispondente ad un nodo del grafo. π_i è solitamente detta il *potenziale* del nodo i. Trasponendo la matrice dei vincoli del primale e ricordando le regole per la costruzione del duale, è immediato trovare il seguente programma:

$$\begin{aligned} \max \quad & (1-n)\pi_r + \sum_{j\neq r} \pi_j \\ \text{s. a} \quad & \pi_j - \pi_i \leq c_{ij} \quad \forall (i,j) \in A \end{aligned} \tag{2.2}$$

In particolare, non vi è alcuna restrizione sulla positività/negatività delle variabili in quanto tutti i vincoli del programma primale sono posti a uguaglianza.
Senza perdita di generalità possiamo assumere che la funzione c ammetta solo valori interi: essendo la matrice dei vincoli totalmente unimodulare, allora anche le variabili π_i della soluzione ottima sono intere per ogni $i \in N$. Per questo motivo non è necessario specificare la condizione $\pi_i \in \mathbb{Z}$ per ogni $i \in N$.
Inoltre, definendo per ogni arco $(i, j) \in A$ il relativo **costo ridotto** $\bar{c}_{ij} = c_{ij} + \pi_i - \pi_j$, i vincoli di ammissibilità duale possono essere riscritti nel modo seguente:

$$\bar{c}_{ij} \geq 0 \quad \forall (i, j) \in A$$

Concludiamo enunciando un criterio di condizioni necessarie e sufficienti per riconoscere un albero di cammini minimi. Questo teorema sta alla base del funzionamento della maggior parte degli algoritmi risolutivi per il problema in questione.
Sia $G = (N, A)$ un grafo orientato fortemente connesso senza cicli di costo negativo. Supponiamo di conoscere T, un albero (orientato) ricoprente con radice r. Chiamiamo $d(i)$ il costo del cammino minimo da r a i, visto come sottografo di T, per ogni $i \in N$.

Teorema 2.1.1 (Condizioni di ottimalità)**.** *T è un albero di cammini minimi di radice r se e solo se*

$$d(i) + c_{ij} \geq d(j) \quad \forall (i, j) \in A$$

Notiamo che un vettore d definito come sopra e che soddisfi le condizioni di ottimalità appena presentate è soluzione delle *equazioni di Bellman*.

2.2 L'algoritmo di Dijkstra

Diverse sono le strategie che si possono adottare per risolvere il problema dell'albero dei cammini minimi. Un primo tentativo, ad esempio, consiste nell'usare la programmazione lineare e applicare il metodo del simplesso al rilassamento lineare di (2.1). Tuttavia ciò si rivela spesso nella pratica alquanto inefficiente. Per questo nel tempo sono stati elaborati diversi algoritmi come alternativa a questo metodo. Tra questi, fondamentale è l'algoritmo di Dijkstra, uno degli algoritmi più frequentemente utilizzati proprio per la sua efficienza.

Consideriamo un grafo orientato $G = (N, A)$ fortemente connesso e una funzione $c\colon A \to \mathbb{Z}^+$ che associa ad ogni arco di A un costo **non negativo** (si noti che in tal caso il problema è limitato e dunque la soluzione ottima esiste sempre). L'algoritmo di Dijkstra si basa sul seguente risultato.

Proposizione 2.2.1. *Sia p_i un cammino di costo minimo da r a i, r e $i \in N$, e sia k un nodo intermedio di tale cammino. Allora il cammino p_k da r a k contenuto in p_i è un cammino minimo da r a k.*

Dimostrazione. Denotiamo con $c(p)$ il costo del cammino p. Supponiamo per assurdo che esista un cammino $\hat{p}_k$ da r a k tale che $c(\hat{p}_k) < c(p_k)$.

Chiamiamo p_{ki} il sotto-cammino di p_i da k ad i, cioè $p_i = (p_k \setminus \{k\},\ p_{ki})$. Allora $\hat{p}_i = (\hat{p}_k \setminus \{k\},\ p_{ki})$ è un cammino da r a i e

$$c(\hat{p}_i) = c(\hat{p}_k) + c(p_{ki}) < c(p_k) + c(p_{ki}) = c(p_i)$$

contro l'ipotesi che p_i sia cammino minimo da r a i. □

Essendo i costi degli archi non negativi, nel seguito ci riferiremo ad essi chiamandoli anche *lunghezze*. L'algoritmo di Dijkstra risolve il problema del cammino minimo servendosi di una funzione $d\colon N \to \mathbb{R}^+ \cup +\infty$, detta *distanza provvisoria*, con le seguenti proprietà:

1. per ogni vertice i tale che $d(i)$ è finita, esiste un cammino da r a i di lunghezza $d(i)$;
2. quando l'algoritmo termina, $d(i)$ è la lunghezza del cammino minimo da r a i (cioè è la *distanza effettiva* da r a i, che denoteremo con $d(i)^*$).

Inizialmente $d(r) = 0$ e $d(i) = +\infty$ per ogni $i \in N$, $i \neq r$.
Durante l'esecuzione dell'algoritmo, ogni vertice si trova in uno dei seguenti stati: *candidato*, *non visitato* oppure *scansionato*. Inizialmente r è l'unico nodo *candidato* mentre tutti i rimanenti sono *non visitati*. L'algoritmo consiste nell'iterare il seguente step di *scansione* finché non restano più nodi candidati:

Scansione. Si selezioni un nodo i *candidato* tale che $d(i)$ sia minima. Si converta i da *candidato* a *scansionato*. Per ogni arco (i, j) tale che $d(i) + c_{ij} < d(j)$, si sostituisca $d(j)$ con $d(i) + c_{ij}$ e si contrassegni j come *candidato*.

Come anticipato, l'algoritmo restituisce solo i costi dei cammini minimi. Per determinare i cammini minimi, ad ogni iterazione teniamo annotata una lista di *predecessori* in questo modo: al momento dell'inizializzazione $p(j)$ non è definito né per r, né per i nodi aventi un valore della distanza provvisoria non finito; durante la scansione aggiorniamo $p(j) = i$ se $d(i) + c_{ij} < d(j)$. Dunque quando l'algoritmo termina, $p(j)$ è il nodo prima di j in qualche cammino minimo da r a j.

Osservazione 2. *Chiamiamo i nodi "candidati" tali perché uno tra loro verrà "eletto" come componente della soluzione nell'iterazione successiva. Un nodo scansionato è il nodo che tra i vari candidati è già stato scelto per comporre l'albero dei cammini minimi.*

Teorema 2.2.1. *L'algoritmo di Dijkstra è corretto.*

Dimostrazione. La dimostrazione è per induzione. L'ipotesi induttiva è che dopo t scansioni $d(j)$ è correttamente calcolata per tutti i nodi j scansionati e inoltre $d(j) = d(j)^*$, cioè la *distanza provvisoria* coincide con la *distanza effettiva* di un cammino minimo da r a j, posto però che tutti i nodi intermedi di tale cammino siano già stati scansionati. Ciò è certamente vero al passo iniziale, poiché r è l'unico nodo scansionato e $d(r) = d(r)^* = 0$. Per ipotesi induttiva, $d(j) \geq d(j)^*$ per tutti i nodi j non scansionati. Poniamoci al $(t+1)$-esimo step e sia i il nodo *candidato* definito nel passo della scansione, cioè il nodo che

minimizzi $d(i)$. Supponiamo per assurdo che $d(i) > d(i)^*$: allora necessariamente il cammino minimo p_i da r ad i trovato deve contenere almeno un nodo non *scansionato*. Sia k il primo di questi nodi. Per la Proposizione 2.2.1, il sotto-cammino p_k di p_i deve essere un cammino minimo. Tuttavia per costruzione tutti i nodi intermedi di p_k sono *scansionati*. Dunque $d(k) = d(k)^* \leq d(i)^* < d(i)$, il che contraddice la scelta di i. Dunque $d(i) = d(i)^*$ e ciò dimostra che il criterio con cui viene scelto i è corretto e che, dopo la scansione, $d(i)^*$ viene impostato definitivamente.
Infine, per dimostrare che, per ogni nodo j *candidato* il cui cammino contenga solo nodi *scansionati*, $d(j)$ rappresenta effettivamente la lunghezza di un cammino minimo da r a j, è sufficiente osservare che il cammino trovato alla $(t+1)$-esima scansione è uguale al precedente o contiene i come ultimo nodo, nel cui caso $d(j) = d(i)^* + c_{ij}$. Per ipotesi induttiva, la tesi è verificata. □

Dalla dimostrazione del Teorema 2.2.1 si deduce un'importante proprietà dell'algoritmo, che enunciamo nella seguente proposizione.

Proposizione 2.2.2. *Durante l'esecuzione dell'algoritmo di Dijkstra ogni nodo $i \in N$ diviene candidato e scansionato esattamente una volta. In particolare, tutti i nodi che vengono contrassegnati come "scansionati" mantengono questa assegnazione definitivamente.*

Infatti ogni volta che un nodo $i \in N$ diventa *scansionato*, si ha che $d(i) = d(i)^*$, il quale corrisponde al minimo valore che $d(i)$ può assumere; dunque la sua assegnazione è definitiva. Gli algoritmi per cui vale questa proprietà vengono spesso chiamati *label-setting methods*. In particolare, diremo che la procedura di Dijkstra funziona in modo **permanente**: questo concetto ritornerà utile in seguito.
La caratteristica appena descritta è importante perché rende l'algoritmo estremamente efficiente: la questione verrà approfondita in seguito parlando di complessità computazionale e metodi di implementazione per l'algoritmo di Dijkstra.

2.2.1 Esempio

Vogliamo determinare i cammini di minima distanza da Chicago ad altre nove città degli Stati Uniti medio-occidentali. Pensiamo ad ogni città come il nodo di un grafo e le indichiamo con $r, a, b, c, \ldots$. Le distanze scritte in Tabella 2.1 sono riportate in miglia/10 e chiaramente $c_{ij} = c_{ji}$ per ogni $i \neq j$.
La Tabella 2.2 mostra le iterazioni dell'algoritmo di Dijkstra e ad ogni passo vengono riportate le coppie $(d(i), p(i))$ per ciascun nodo i. L'iterazione 0 coincide con l'inizializzazione dell'algoritmo; negli step successivi, se un valore di d è evidenziato in grassetto, allora ciò indica che il corrispondente nodo è stato convertito in *scansionato* e dunque d è stato assegnato in modo definitivo.

		a	b	c	d	e	f	g	h	i
r.	Chicago	96	105	50	41	86	46	29	56	70
a.	Dallas		78	49	94	21	64	63	41	37
b.	Denver			60	84	61	54	86	76	51
c.	Kansas City (MO)				45	35	20	26	17	18
d.	Minneapolis					80	36	55	59	64
e.	Oklahoma City						46	50	28	8
f.	Omaha							45	37	30
g.	St. Louis								21	45
h.	Springfield (MO)									25
i.	Wichita									

Tabella 2.1: Distanze relative tra le città in miglia/10.

Iterazione	r	a	b	c	d	e	f	g	h	i
0	$0, -$	$\infty, -$	$\infty, -$	$\infty, -$	$\infty, -$	$\infty, -$	$\infty, -$	$\infty, -$	$\infty, -$	$\infty, -$
1	$\mathbf{0}, -$	$96, r$	$105, r$	$50, r$	$41, r$	$86, r$	$46, r$	$29, r$	$56, r$	$70, r$
2		$92, g$	$105, r$	$50, r$	$41, r$	$79, g$	$46, r$	$\mathbf{29}, \mathbf{r}$	$50, g$	$70, r$
3		$92, g$	$105, r$	$50, r$	$\mathbf{41}, \mathbf{r}$	$79, g$	$46, r$		$50, g$	$70, r$
4		$92, g$	$100, f$	$50, r$		$79, g$	$\mathbf{46}, \mathbf{r}$		$50, g$	$70, r$
5		$92, g$	$100, f$	$\mathbf{50}, \mathbf{r}$		$79, g$			$50, g$	$68, c$
6		$91, h$	$100, f$			$78, h$			$\mathbf{50}, \mathbf{g}$	$68, c$
7		$91, h$	$100, f$			$76, i$				$\mathbf{68}, \mathbf{c}$
8		$91, h$	$100, f$			$\mathbf{76}, \mathbf{i}$				
9		$\mathbf{91}, \mathbf{h}$	$100, f$							
10			$\mathbf{100}, \mathbf{f}$							

Tabella 2.2: Iterazioni dell'algoritmo di Dijkstra.

Si osserva facilmente che l'algoritmo di Dijkstra può non funzionare correttamente se il grafo contiene archi con costi negativi. Un semplice esempio è mostrato in Figura 2.1. Alla prima iterazione l'algoritmo assegnerebbe permanentemente il valore $d(b) = d(b)^* = 3$, mentre $d(b)^* = d(a) + c_{ab} = 2$. In particolare, preso un nodo *candidato*, non è più valido impostare $d(i) = d(i)^*$ solo perché $d(i)$ è il minimo valore degli $d(j)$ tra tutti i nodi *candidati* j.

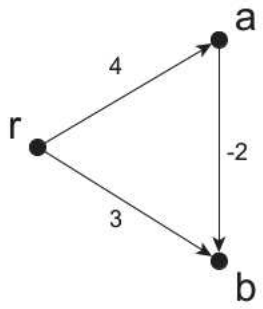

Figura 2.1: Grafo per cui l'algoritmo di Dijkstra fallisce.

Tuttavia, se il grafo non contiene alcun ciclo di costo negativo, l'algoritmo si può modificare per trattare correttamente archi con costi negativi. La modifica essenziale è che nessuno dei valori $d(j)$ viene posto uguale a $d(j)^*$ finché non sono state eseguite m iterazioni. In questo modo si ottiene l'**algoritmo di Bellman-Ford** in cui, ad ogni iterazione, si controllano le condizioni di ottimalità per tutti i nodi del grafo e, per l'appunto, solo quando l'algoritmo giunge a terminazione si ottengono i cammini minimi per tutti i nodi. Per un'accurata descrizione della procedura si faccia riferimento a [9].

2.3 Implementazione e costo computazionale

Prima di parlare del costo computazionale dell'algoritmo di Dijkstra, facciamo qualche considerazione generale.

Come notato in precedenza nel Teorema 2.1.1, le *equazioni di Bellman* sono strettamente legate all'ottimalità delle soluzioni. Di fatto, durante l'esecuzione dell'algoritmo di Dijkstra, esse vengono continuamente controllate per individuare i nodi candidati a formare la soluzione ottima.
Più in generale, la maggior parte degli algoritmi che risolvono problemi di cammino minimo condividono questo comportamento. In realtà, molti di questi algoritmi presentano una struttura ricorrente, nel senso che eseguono le seguenti operazioni:

1. si inizializza un albero T con radice r e per ogni nodo $i \in N$ si definisce una *distanza provvisoria* $d(i)$ che rappresenta la lunghezza di un qualche cammino da r a i in T;

2. identificato un arco $(i, j) \in A$ tale che valga $d(i) + c_{ij} < d(j)$, si modifica il vettore d ponendo $d(j) = d(i) + c_{ij}$ e si aggiorna l'albero T sostituendo l'arco attualmente incidente con il nuovo arco (i, j);

3. si ripete il passo 2 finchè le condizioni di ottimalità del Teorema 2.1.1 sono soddisfatte.

Veniamo ora alla questione di come l'implementazione di una procedura sia legata al suo costo computazionale. Dei tre passi sopra elencati, un punto cruciale è il modo in cui selezionare gli archi al **passo 2**, cioè quelli che non soddisfano le condizioni di ottimalità. Diverse implementazioni di questo passaggio portano a diverse varianti dello stesso algoritmo. Per di più, non è un caso che il comportamento di una procedura sia fortemente influenzato dal modo in cui questa operazione è svolta. Ecco che, come conseguenza, il costo computazionale e quindi l'efficienza di un algoritmo dipendono in maggior parte dalle scelte che facciamo in fase di implementazione, cioè le scelte riguardanti le *regole di selezione* e le *strutture dati* per il nostro programma.

Solitamente si procede in questo modo: siccome $n \leq m$ è ragionevole selezionare i nodi anziché gli archi e, una volta scelto un nodo i, svolgere le operazioni del passo 2 su $FS(i)$. Definiamo allora un insieme Q contenente i nodi candidati a tale selezione. Il problema al passo 2 si riformula così: come selezionare un nodo i dall'insieme di candidati Q?

Ora le possibilità sono davvero numerose, poiché si entra nell'ambito delle *regole di selezione*: ognuna di queste regole induce una particolare strategia di ricerca su G. Ad esempio la ricerca in ampiezza (*breadth-first search*) o la ricerca in profondità (*depth-first search*) portano ad algoritmi di "ricerca su lista", mentre la ricerca di minimo (*lowest-first search*) porta ad algoritmi in cui si ricerca un nodo con una "proprietà minima", come per l'appunto l'algoritmo di Dijkstra.

Ora, per quanto riguarda l'algoritmo di Dijkstra nello specifico, si può stimare in modo un po' grossolano il suo costo computazione nel seguente modo: sia Q l'insieme dei nodi *candidati*. Ad ogni passo dell'algoritmo, l'insieme del nodi *scansionati* aumenta di un elemento. Quando $|Q| = k$, $1 \leq k \leq n - 1$, la scansione richiede di trovare il minimo

tra k elementi e svolge non più di k addizioni e confronti. Dunque il massimo numero di operazioni base svolte è limitato da $c\sum_{k=1}^{n-1} k$ per qualche costante c. Dunque l'algoritmo è $O(n^2)$.

Questa stima tuttavia non è ottimale. La strategia per migliorarla è servirsi di apposite strutture dati più sofisticate. Nel nostro contesto, ne citiamo due che rendono l'implementazione degna di nota, ciascuna per un proprio motivo. La prima, ampiamente descritta in [3], usa la struttura dati dei *Fibonacci heaps* (*F-heaps*) per implementare l'insieme Q. Così facendo, è possibile dimostrare che la procedura di Dijkstra richiede un costo compoutazionale pari a $O(m + n \log n)$. Di fatto sappiamo che questo è il miglior costo possibile per un algoritmo fortemente polinomiale che risolve il problema dei cammini minimi nel caso in cui non vi siano archi di costo negativo. In particolare, si tratta dell'implementazione dell'algoritmo di Dijkstra con miglior costo fortemente polinomiale.

Il secondo metodo, proposto per la prima volta da R. Dial e descritto in [5] e [6], fa uso di una struttura dati *one-level buckets*. L'importanza di questa strategia risiede nel fatto che è possibile stabilire una relazione tra costo computazionale e i costi effettivi assegnati agli archi del grafo in esame. Infatti, se definiamo $c_{\max}$ come il massimo di tali costi, cioè

$$c_{\max} = \max\{c_{ij} \colon (i,j) \in A\}$$

è possibile dimostrare che l'algoritmo di Dijkstra con questa implementazione calcola i cammini minimi in un tempo che è pari a $O(m + nc_{\max})$. In particolare, se accade che gli archi presentino dei costi i cui valori siano sufficientemente "piccoli", allora con questa strategia si può ottenere un notevole guadagno in termini di costo computazionale. Poiché l'implementazione di Dial risulterà significativa più avanti, segue ora una presentazione sintetica della sua struttura.

2.3.1 Implementazione di Dial dell'algoritmo di Dijkstra

Come già anticipato, l'operazione più dispendiosa, in termini computazionali, eseguita dall'algoritmo di Dijkstra è la selezione, dall'insieme dei nodi candidati Q, di un vertice avente distanza provvisoria minima. Viene cioè richiesto di identificare un vertice u tale che $d(u) = \min\{d(j) \colon j \in Q\}$. Nell'implementazione proposta da R. Dial il nodo i viene identificato sfruttando la seguente osservazione.

Supponiamo che, durante una qualche iterazione dell'algoritmo di Dijkstra successiva alla prima, l'insieme Q sia non vuoto (altrimenti la procedura termina). Chiaramente, per ogni vertice $j \in Q$, si ha che $d(j) = d(i) + c_{ij}$ per qualche nodo i precedentemente scansionato. Definiamo $c_{\max} = \max\{c_{ij} \colon (i,j) \in A\}$, cioè il valore massimo tra tutti i costi assegnati agli archi del grafo. Osserviamo che, se v è l'ultimo nodo che è stato scansionato, allora vale che

$$d(v)^* \leq d(j) \leq d(v)^* + c_{\max} \qquad \forall j \in Q \tag{2.3}$$

La disuguaglianza di sinistra in (2.3) è giustificata dal fatto che l'algoritmo di Dijkstra assegna ai nodi i valori di d^* in ordine crescente.

Le disequazioni in (2.3) implicano che ad ogni iterazione tutti i valori calcolati dall'algoritmo appartengono ad un intervallo di ampiezza fissata pari a $(c_{\max} + 1)$. Ciò significa

che è possibile rappresentare modulo $(c_{\max}+1)$ tutti gli interi $d(j)$, per ogni $j \in Q$, in modo unico. L'idea di Dial consiste nell'utilizzare una specifica struttura dati, quella dei *one-level buckets*, per immagazzinare tutti i valori di d durante ciascuna iterazione dell'algoritmo.
Un *bucket* è una struttura dati indicizzata da una chiave che può contenere una lista di elementi. In base alle osservazioni fatte precedentemente, nel nostro caso sarà sufficiente utilizzare $(c_{\max}+1)$ buckets: per ciascuno di essi la chiave è rappresentata da un intero appartenente all'intervallo $[0, c_{\max}]$. Dunque i buckets formano un insieme totalmente ordinato, dove l'ordine è dato dalla sequenzialità delle chiavi. Infine ciascun bucket contiene una lista di nodi che corrisponde all'insieme dei vertici che presentano una distanza provvisoria pari alla chiave del bucket, modulo $(c_{\max}+1)$. In altre parole, ad ogni iterazione della procedura, il k-esimo bucket, con k intero e $k \in [0, c_{\max}]$, contiene tutti quei nodi i tali che $d(i) \equiv k \mod c_{\max}+1$.

La selezione di u, il nodo avente distanza provvisoria minima, avviene nel seguente modo. Sia v l'ultimo nodo scansionato dall'algoritmo, con $d(v) \equiv k \mod c_{\max}+1$. A partire dal k-esimo bucket, i buckets vengono esaminati sequenzialmente tramite le loro chiavi fino a che ne viene trovato uno che sia non vuoto. Sia d l'indice corrispondente (cioè il primo bucket non vuoto è il d-esimo). Allora u è uno qualunque dei nodi appartenenti alla lista contenuta nel d-esimo bucket. Se l'indice c_{max} viene raggiunto senza trovare alcun bucket non vuoto, la ricerca riprende sequenzialmente dall'indice 0. Una volta identificato il nodo u, si procede con le altre operazioni dell'algoritmo di Dijkstra. In particolare, ogniqualvolta venga ridefinito, per un certo nodo i, il valore $d(i)$, allora il nodo i deve essere riassegnato e spostato nel bucket opportuno. L'algoritmo di Dijkstra termina quando, durante la ricerca sequenziale dei buckets, tutti gli indici vengono esaminati e nessun nodo trovato (corrisponde al caso in cui tutti i buckets sono vuoti ed equivalentemente $Q = \emptyset$).
Si noti che con questa strategia non è mai necessario eseguire più di $c_{\max}+1$ confronti durante ciascuna iterazione, rendendo di fatto la parte più costosa dell'algoritmo relativamente efficiente. Ne segue che la ricerca complessiva dei nodi di minima distanza nella procedura di Dijkstra prevede un costo computazionale pari a $O(nc_{\max})$. Se a questo dato aggiungiamo il costo necessario per le altre operazioni, allora l'intera procedura di Dijkstra richiede non più di $O(m + nc_{\max})$ operazioni.
Concludiamo con un'ultima osservazione. Sia w l'ultimo nodo scansionato dalla procedura di Dijkstra: il cammino da r a w è, tra tutti i cammini minimi della soluzione corrente, quello con costo maggiore, pari a $d(w)^*$. In linea teorica, è possibile ridurre ulteriormente il numero di buckets da $(c_{\max}+1)$ a C, dove C è una qualunque stima dall'alto di $d(w)^*$. In tal caso, come prevedibile, il numero di confronti necessari ad ogni iterazione si riduce a C. Questa proprietà verrà più volte sfruttata nel seguito nelle procedure di riottimizzazione.

Capitolo 3

Riottimizzare cammini minimi

3.1 Tecniche di riottimizzazione

In Ricerca Operativa, in ambiti in cui si usano i grafi per creare modelli, è frequente che ci si trovi a dover risolvere una sequenza di problemi di cammino minimo in cui due successive istanze differiscono tra di loro per un piccolo cambiamento nella struttura del grafo: tale modifica può riguardare l'insieme dei nodi, l'insieme degli archi oppure entrambi. Contesti in cui ciò accade sono ad esempio quelli riguardanti trasporti su larga scala, instradamento di veicoli o particolari modelli comunicativi.
Chiaramente un naturale approccio risolutivo consiste nel risolvere ciascun problema ex novo con un algoritmo per l'albero dei cammini minimi, indipendentemente dal caso precedente. Ciò però non è l'unica possibilità: per questo si studiano le strategie di riottimizzazione.
La *riottimizzazione* consiste nell'affrontare il k-esimo problema di cammino minimo della sequenza riutilizzando informazioni utili provenienti dalla soluzione del (k-1)-esimo. L'obiettivo di queste tecniche sta nel ridurre il carico computazionale richiesto rispetto alla risoluzione ex novo. La strategia di riottimizzazione presentata, formulata da S. Pallottino e M. G. Scutellà in [8], affronta il caso più generale, in cui un qualunque sottoinsieme di archi del grafo è soggetto ad un cambio di costo, sia esso in aumento o in diminuzione.

Nel contesto appena descritto, sia $G = (N, A)$ un grafo orientato fortemente connesso ai cui archi sono associati dei costi non negativi. Supponiamo di conoscere $T_r^* = (N, A_r)$ albero dei cammini minimi con radice r e sia $\pi^{(r)}$ il corrispondente vettore duale ottimo avente componenti $\pi_i^{(r)}$, $i \in N$. Assumeremo d'ora in avanti che i costi c_{ij} siano interi, il che comporta l'interezza delle componenti di $\pi^{(r)}$ per unimodularità della matrice dei vincoli del relativo programma lineare. Con la notazione introdotta in precedenza per i costi ridotti $\bar{c}_{ij} = c_{ij} + \pi_i^{(r)} - \pi_j^{(r)}$, si ha che l'ammissibilità duale si traduce in

$$\bar{c}_{ij} \geq 0 \quad \forall (i,j) \in A \tag{3.1}$$

Osserviamo che, in base a come sono state definite le variabili primali x_{ij} si ha che $x_{ij} \neq 0$ per ogni $(i,j) \in A_r$ perché chiaramente almeno un cammino minimo passa per ogni arco della soluzione. Per il Teorema degli Scarti Complementari, questo implica la seguente

condizione:

$$\bar{c}_{ij} = 0 \quad \forall (i,j) \in A_r \tag{3.2}$$

Assumiamo ora che ad alcuni degli archi del grafo vengano assegnati dei nuovi costi, sia maggiori che minori rispetto ai costi precedenti. Chiamiamo K tale sottoinsieme di archi. Denotiamo poi con c'_{ij} i nuovi costi assegnati agli archi $(i,j) \in A$, ponendo $c'_{ij} = c_{ij}$ se $(i,j) \notin K$. Definiamo infine i cosiddetti **costi ridotti modificati**, cioè le quantità

$$\bar{c}'_{ij} = c'_{ij} + \pi_i^{(r)} - \pi_j^{(r)}$$

Dalla (3.1) si ha che, per ogni $(i,j) \in A \setminus K$, $\bar{c}'_{ij} = \bar{c}_{ij} \geq 0$, cioè l'ammissibilità duale di $\pi^{(r)}$ viene preservata per i vincoli corrispondenti agli archi che non cambiano costo. In particolare, $\bar{c}'_{ij} = \bar{c}_{ij} = 0$ per ogni $(i,j) \in A_r \setminus K$. Nulla invece si può dire, per il momento, sul segno di $\bar{c}'_{ij}$ per $(i,j) \in K$. Queste proprietà verranno sfruttate per riottimizzare T_r^*.

La strategia di riottimizzazione è la seguente: l'insieme K viene suddiviso in sottoinsiemi disgiunti $K_1, \ldots, K_s$ e vengono poi eseguite s fasi in sequenza, una per ciascuno di questi sottoinsiemi. Alla i-esima fase T_r^* viene riottimizzato rispetto ai nuovi costi degli archi di K_i. Più precisamente, ogni fase riceve in input l'attuale albero ottimo T_r^* e l'attuale vettore duale ottimo $\pi^{(r)}$, modifica i costi degli archi di un sottoinsieme K_i e, tramite specifiche operazioni, restituisce una versione aggiornata delle soluzioni, che verranno poi usate come input della fase successiva. Al termine dell'ultima fase, il risultato è la soluzione ottima del problema in cui il costo di tutti gli archi di K è stato modificato.

Tale approccio dipende notevolmente dalla scelta della partizione di K. Nel seguito, consideriamo prima un caso semplice in cui ogni sottoinsieme di K contiene un solo arco, poi un caso in cui la scelta è un po' più sofisticata.

3.2 Decomposizione di K in singoli archi

Consideriamo come suddivisione di K una partizione in singoli archi. Sia $T_r^* = (N, A_r)$ l'attuale albero di cammini minimi, soluzione ottima del primale, $\pi^{(r)}$ il vettore duale ottimo e siano c_{ij} i costi relativi agli archi $(i,j) \in A$. Selezioniamo un arco $(u,v) \in K$ e modifichiamo il suo costo, ignorando i nuovi costi degli altri archi di K.
Per semplicità di notazione indichiamo per il momento con T_r l'albero ottimo T_r^* e, per ogni nodo $i \in N$, sia $d(i)^*$ la lunghezza di un cammino minimo da r a i contenuto in T_r (si noti che tale distanza è la stessa che viene calcolata alla fine di una esecuzione dell'algoritmo di Dijkstra). Chiamiamo poi T'_r la soluzione ottima ricercata, cioè quella che tiene conto del nuovo costo c'_{uv} e sia $d(i)'^*$ la distanza minima da r al nodo i in T'_r, similmente a come definito sopra. Definiamo poi l'insieme

$$S(u) = \{j \in N \colon d(j)^* \leq d(u)^*\}$$

cioè l'insieme di nodi che hanno distanza minima da r minore o uguale a $d(u)$, sempre rispetto a T_r. Infine indichiamo con $T_r(v)$ il sotto-albero di T_r contenente v e tutti i suoi nodi e archi che da v discendono. Raccogliamo in un unico teorema i risultati fondamentali riguardanti la relazione tra T_r e T'_r.

Teorema 3.2.1. *Sia $c'_{uv} > 0$ il nuovo costo dell'arco (u, v). Valgono le seguenti affermazioni:*

1. *Se $c'_{uv} < c_{uv}$ e $(u, v) \in T_r$, allora l'ottimalità viene preservata per gli archi di $T_r(v)$ e per essi si ricavano esplicitamente i costi dei cammini minimi che compongono. Restano ottimi anche i cammini che terminano in nodi appartenenti ad $S(u)$:*

$$\begin{aligned} &T_r(v) \subseteq T'_r(v) \\ &j \in T_r(v) \Rightarrow d(j)'^* = d(j)^* - (c_{uv} - c'_{uv}) \\ &j \in S(u) \Rightarrow d(j)'^* = d(j)^* \end{aligned}$$

2. *Se $c'_{uv} < c_{uv}$ e $(u, v) \notin T_r$, allora continuano a rimanere ottimi gli archi appartenenti a $T_r(v)$ e i cammini terminanti in nodi di $S(u)$:*

$$\begin{aligned} &T_r(v) \subseteq T'_r(v) \\ &j \in S(u) \Rightarrow d(j)'^* = d(j)^* \end{aligned}$$

3. *Se $c'_{uv} > c_{uv}$ e $(u, v) \in T_r$, allora gli archi non appartenenti a $T_r(v)$ restano ottimi:*

$$j \notin T_r(v) \Rightarrow d(j)'^* = d(j)^*$$

4. *Se $c'_{uv} > c_{uv}$ e $(u, v) \notin T_r$, allora la soluzione precedentemente calcolata resta ottima:*

$$\begin{aligned} &T'_r = T_r \\ &d(j)'^* = d(j)^* \quad \forall j \in N \end{aligned}$$

Dimostrazione. (1). Siccome l'arco (u, v) è presente al massimo una volta in un cammino minimo, deve accadere che

$$d(j)'^* \geq d(j)^* - (c_{uv} - c'_{uv}) \quad \forall j \in N \tag{3.3}$$

Dato che per ogni nodo $j \in T_r(v)$ esiste un cammino che ha esattamente il costo $d(j)^* - (c_{uv} - c'_{uv})$, cioè proprio il sotto-cammino di T_r da r a j con il nuovo costo c_{uv}, allora per tali nodi la condizione (3.3) viene soddisfatta ad uguaglianza. Ne segue che per essi il cammino minimo non cambia e dunque $T_r(v) \subseteq T'_r(v)$.
Per quanto riguarda la seconda affermazione, osserviamo che tutti i cammini minimi che terminano in nodi di $S(u)$ sicuramente non contenevano l'arco (u, v) prima del cambio di costo. Dopo il cambio, ogni cammino contenente l'arco (u, v) ha lunghezza non minore di $d(u)^*$ (poiché $c'_{uv} > 0$). Per questo motivo per tali nodi la distanza minima dalla radice non cambia.
(2). Dopo il cambio di costo, l'arco (u, v) potrebbe entrare oppure non entrare nell'albero di cammini minimi. Per ogni nodo $j \in T_r(v)$, nel primo caso la distanza minima da r diminuisce della quantità $c_{uv} - c'_{uv}$, mentre nel secondo essa non cambia. In entrambi i casi $T_r(v)$ rimane nell'albero di cammini minimi. La dimostrazione della seconda affermazione è identica a quella presentata in (1).

(3). Per ogni vertice $j \notin T_r(v)$ il cammino minimo che termina in j non conteneva l'arco (u, v) prima del cambio di costo. A maggior ragione, tale cammino non conterrà (u, v) dopo la sostituzione di c_{uv} con c'_{uv}. Dunque per tali archi le distanze minime restano immutate.
(4). Segue immediatamente dalle ipotesi. □

Alla luce dei risultati di questo teorema, possiamo pensare ad un algoritmo di riottimizzazione per il cambio di costo di un arco (u, v). Analizziamo i due casi distinti che si ottengono a seconda che il costo di (u, v) aumenti oppure diminuisca.

3.2.1 Aumento del costo di un arco

Per quanto dimostrato nel teorema precedente, se $(u, v) \notin T_r^*$ allora la soluzione corrente resta ottima. Se invece si ha che $(u, v) \in T_r^*$, la strategia che adottiamo è la seguente. Calcoliamo i costi ridotti modificati per ciascun arco del grafo:

$$\bar{c}'_{ij} = c'_{ij} + \pi_i^{(r)} - \pi_j^{(r)} \quad \forall (i, j) \in A \tag{3.4}$$

dove π_i e π_j sono componenti del vettore $\pi^{(r)}$; ricordiamo che $c'_{ij} = c_{ij}$ per ogni $(i, j) \neq (u, v)$. Chiamiamo ora una procedura di Dijkstra completa sul grafo, associando però agli archi non i costi effettivi, bensì tali costi modificati. Tale algoritmo effettuerà comunque $n = |N|$ iterazioni, come per una chiamata standard di Dijkstra, ma con un vantaggio: siccome ogni valore $d(i)$ generato dall'algoritmo è un numero intero appartenente all'intervallo $[0, \bar{c}'_{uv}]$, è possibile eseguire la riottimizzazione in un tempo di $O(m+\min\{n \log n, C\})$, dove $C = \bar{c}'_{uv}$.
In altre parole, un discreto margine di miglioramento per la riottimizzazione può essere ottenuto sfruttando l'implementazione di Dial a one-level buckets, che in questo caso si rivela fondamentale poiché rende tale metodo, in linea di massima, preferibile rispetto alla risoluzione ex novo. In particolare, essa risulta conveniente in caso di "piccole perturbazioni" di costo.

La correttezza di questo algoritmo può essere dimostrata osservando nel dettaglio l'equazione (3.4). In particolare, vale la seguente importante proposizione riguardante le trasformazioni lineari per i costi degli archi.

Proposizione 3.2.1. *Il problema dell'albero dei cammini minimi che si ottiene assegnando agli archi i costi c_{ij} è equivalente al problema in cui i costi degli archi sono dati da:*

$$\bar{c}_{ij} = c_{ij} + \pi_i - \pi_j \quad \forall (i, j) \in A$$

dove $\pi_1, \ldots, \pi_n$ sono interi tali che $\bar{c}_{ij} \geq 0$ per ogni $(i, j) \in A$.

Dimostrazione. Siano r e s due nodi del grafo e chiamiamo P un qualunque cammino da r a s. Definiamo $c(P)$ come il costo del cammino P quando associamo agli archi i costi c_{ij}, mentre chiamiamo $\bar{c}(P)$ il costo dello stesso cammino P se consideriamo come costi le quantità $\bar{c}_{ij}$. Troviamo che:

$$\bar{c}(P) = \sum_{(i,j)\in P} \bar{c}_{ij} = \sum_{(i,j)\in P} (c_{ij} + \pi_i - \pi_j) = c(P) + \pi_r - \pi_s$$

Abbiamo dunque che $\bar{c}(P)$ e $c(P)$ differiscono per una costante, la quale è indipendente dal particolare cammino scelto per giungere da r a s. Di conseguenza, chiamando S l'insieme di tutti cammini da r a s,

$$\min_{P \in S} \bar{c}(P) = \min_{P \in S} \{c(P) + \pi_r - \pi_s\} = \min_{P \in S} \{c(P)\} + \pi_r - \pi_s$$

Dunque se un cammino è minimo per i costi c_{ij}, allora è minimo anche per i costi $\bar{c}_{ij}$ e viceversa. □

È importante notare che questa non è l'unica strada che si può percorrere. Si osservi ad esempio che, essendo $\bar{c}'_{uv} \geq 0$, dopo il cambio di costo **l'ammissibilità duale è preservata** per il vettore $\pi^{(r)}$. Questo fatto è degno di nota perché per questo tipo di situazioni sono stati sviluppati degli algoritmi detti "duali" che funzionano in questo modo: partendo da una soluzione duale ammissibile, si cerca una nuova soluzione che mantenga l'ammissibilità duale ma che garantisca una diminuzione di costo della soluzione primale. Si continua così fino a trovare una soluzione ammissibile anche nel primale che rispetti le condizioni degli Scarti Complementari. A tal punto l'algoritmo termina perché la soluzione trovata è ottimale.
Nel caso del problema dell'albero dei cammini minimi, il procedimento è il seguente: data una soluzione duale ammissibile π e un albero di cammini minimi parziale T, l'algoritmo aumenta T, passo dopo passo, cercando archi che soddisfano le condizioni degli Scarti Complementari rispetto a π. Questi algoritmi costituiscono una valida alternativa a quello proposto e vengono ampiamente descritti in [7].

3.2.2 Diminuzione del costo di un arco

Supponiamo ora che il costo dell'arco (u, v) diminuisca. Sia $c'_{uv} < c_{uv}$ il nuovo costo e $\bar{c}'_{uv} = c'_{uv} + \pi_u^{(r)} - \pi_v^{(r)}$ il corrispondente costo ridotto. Per applicare la seguente strategia, supponiamo che la diminuzione di costo sia tale che $\bar{c}'_{uv} < 0$. Ricordiamo che per ogni altro arco $(i, j) \neq (u, v)$ si ha che $\bar{c}'_{ij} = \bar{c}_{ij} \geq 0$.
In base ai risultati del Teorema 3.2.1. presentiamo una procedura di riottimizzazione che fa uso nuovamente dei costi ridotti modificati $\bar{c}'_{ij}$ come costi degli archi.

Procedura di riottimizzazione

1. Si inizializza il valore $d(i)$ a zero per ciascun nodo $i \neq v$ del grafo. Si imposta invece $d(v) = \bar{c}'_{uv}$. Si predispone u come predecessore di v mantenendo come predecessori degli altri nodi quelli indicati dalla soluzione T_r^*. Si inizializzi l'insieme Q dei nodi candidati con il solo nodo v: $Q = \{v\}$.

2. Si chiami l'algoritmo di Dijkstra sull'insieme Q precedentemente inizializzato. Ad ogni iterazione della procedura, l'algoritmo estrae da Q il nodo i tale che $d(i)$ sia minimo tra tutti i nodi di Q e controlla l'ottimalità di ogni arco $(i, j) \in FS(i)$ attraverso le equazioni di Bellman: se $d(i) + \bar{c}'_{ij} < d(j)$, si sostituisce $d(j)$ con $d(i) + c_{ij}$ e si aggiunge j all'insieme Q.

In questa procedura ogni nodo che viene inserito in Q ha un valore di d negativo. In particolare, si può affermare che ogni valore $d(j)$ calcolato dall'algoritmo di Dijkstra misura la contrazione del cammino minimo da r a j dovuta alla riduzione di costo di (u, v) rispetto alla soluzione precedente. Questo concetto verrà approfondito nel successivo esempio; tuttavia, come prova di questa affermazione, si noti che, per ogni nodo $i \in T_r(v)$, l'algoritmo di Dijkstra riduce il valore di $d(i)$ della quantità $\bar{c}'_{uv}$. Non solo, anche vertici al di fuori di $T_r(v)$ possono subire la stessa riduzione di costo. Ciò tuttavia non avviene per i nodi lungo il cammino che congiunge r a v in T_r^*, poiché ciò creerebbe un ciclo di costo negativo.

Al termine della procedura il valore del vettore duale $\pi^{(r)}$ può essere aggiornato nel seguente modo:

$$\pi_i^{(r)} := \pi_i^{(r)} + d(i) \quad \forall i \in N$$

il quale nuovamente ha componenti intere siccome sono interi i vettori al membro di destra. Nello specifico, $\pi_i^{(r)}$ deve essere aggiornato solo per i nodi i aventi un valore di $d(i)$ negativo, cioè quelli che sono stati inseriti in Q. Il nuovo albero di cammini minimi aggiornato viene restituito dalla procedura sopra descritta attraverso il vettore dei predecessori $p = [p_i]$.

Un'ultima importante considerazione riguarda la correttezza di tale procedura. In particolare sappiamo che l'algoritmo di Dijkstra può non funzionare in modo corretto se il grafo che stiamo considerando contiene degli archi aventi costo negativo. Nella situazione sopra descritta, il grafo in questione contiene esattamente un arco con costo negativo: infatti $\bar{c}'_{uv} < 0$. È lecito dunque utilizzare l'algoritmo di Dijkstra?
La risposta è affermativa, per la seguente motivazione. Supponiamo che $\bar{c}'_{uv} = c'_{uv} + \pi_u - \pi_v = -\alpha$. Ciò significa che, dopo il cambio di costo, se decidiamo che l'arco (u, v) appartenga alla nuova soluzione, allora il costo del cammino minimo da r a v diminuirà esattamente di α rispetto al costo della soluzione precedente. Siccome questo è il valore minimo che tale costo può assumere, deduciamo che (u, v) dovrà sempre far parte dell'albero dei cammini minimi riottimizzato, indipendentemente dai costi degli altri archi.
Durante la riottimizzazione, il primo nodo a essere scansionato è v: al primo passo vengono dunque assegnati in modo definitivo il valore $d(v)^* = \bar{c}'_{uv}$ e il predecessore $p(v) = u$. In questo modo l'arco (u, v) non verrà mai più considerato nei passi successivi: siccome tutti gli altri archi del grafo hanno costo ridotto non negativo, possiamo concludere che la procedura funziona correttamente.

3.2.3 Esempio

Il seguente grafo (Figura 3.1) contiene 13 nodi, denominati $a, b, c, \ldots, l$ e r, e a ciascun arco è stato assegnato un costo intero. Avviamo la procedura di Dijkstra per capire quali siano i cammini minimi da r a tutti gli altri nodi.
Dopo 12 iterazioni l'algoritmo termina. Ripercorrendo la lista dei predecessori, possiamo costruire l'albero dei cammini minimi, evidenziato in Figura 3.2.

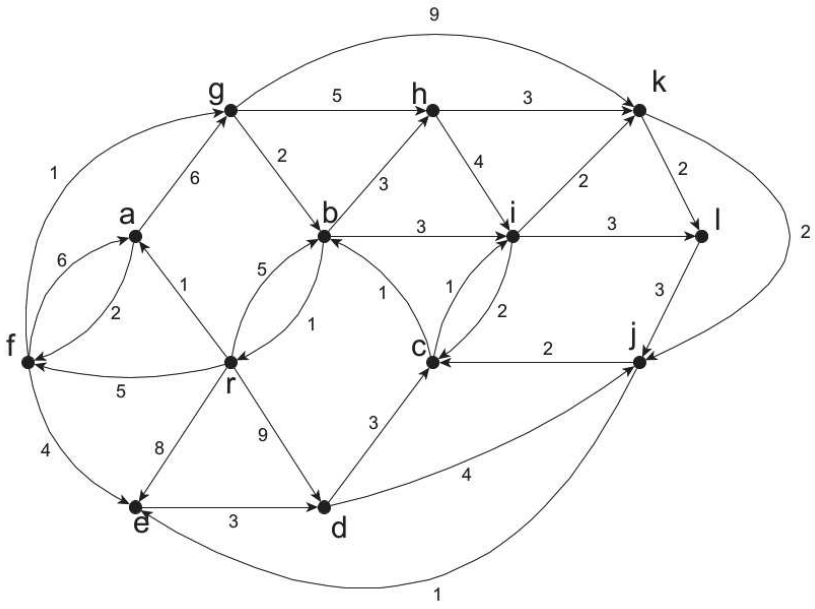

Figura 3.1: Esempio di grafo con costi degli archi non negativi. Si può verificare che il grafo è fortemente connesso.

Iter	r	a	b	c	d	e	f	g	h	i	j	k	l
0	$0,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$
1	$0,-$	$1,r$	$5,r$	$\infty,-$	$9,r$	$8,r$	$5,r$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$
2		$1,r$	$5,r$	$\infty,-$	$9,r$	$8,r$	$3,a$	$7,a$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$
3			$5,r$	$\infty,-$	$9,r$	$7,f$	$3,a$	$4,f$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$
4			$5,r$	$\infty,-$	$9,r$	$7,f$		$4,f$	$9,g$	$\infty,-$	$\infty,-$	$13,g$	$\infty,-$
5			$5,r$	$\infty,-$	$9,r$	$7,f$			$8,b$	$8,b$	$\infty,-$	$13,g$	$\infty,-$
6				$\infty,-$	$9,r$	$7,f$			$8,b$	$8,b$	$\infty,-$	$13,g$	$\infty,-$
7				$\infty,-$	$9,r$				$8,b$	$8,b$	$\infty,-$	$11,h$	$\infty,-$
8				$10,i$	$9,r$					$8,b$	$\infty,-$	$10,i$	$11,i$
9				$10,i$	$9,r$						$13,d$	$10,i$	$11,i$
10				$10,i$							$13,d$	$10,i$	$11,i$
11											$12,k$	$10,i$	$11,i$
12											$12,k$		$11,i$
13											$12,k$		

Tabella 3.1: Iterazioni dell'algoritmo di Dijkstra per trovare i cammini minimi. Ad ogni passo è immediato riconoscere i nodi candidati: essi sono i nodi v tali che $d(v) < \infty$.

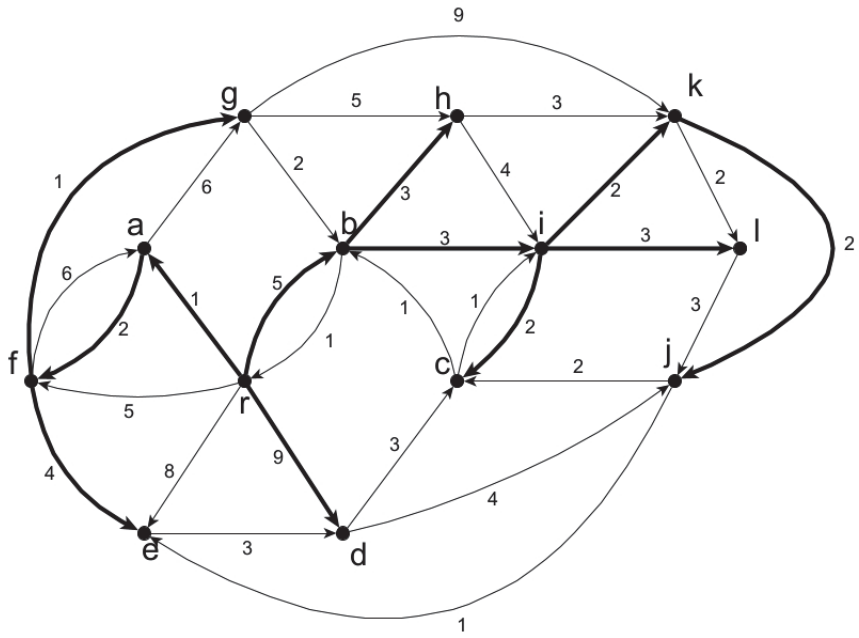

Figura 3.2: Albero dei cammini minimi.

Supponiamo ora che cambi il costo di un arco e riottimizziamo la soluzione con il metodo sopra presentato. In particolare, consideriamo il caso in cui il costo dell'arco (g, k) diminuisce, passando da $c_{gk} = 9$ a $c'_{gk} = 1$.
Con riferimento al Teorema 3.2.1, possiamo fin da subito ottenere delle importanti informazioni. Utilizzando la terminologia precedentemente introdotta, se indichiamo con N l'insieme dei nodi in questo grafo, abbiamo che

$$S(g) = \{j \in N : d(j) \leq d(g)\} = \{j \in N : d(j) \leq 4\} = \{r, a, f, g\}$$
$$T_r(k) = \{k, (k, j), j\}$$

Poiché l'arco (g, k) non appartiene alla soluzione ottima corrente, ci troviamo nel caso (2) del suddetto teorema. Come conseguenza, ci aspettiamo che dopo la riottimizzazione l'arco (k, j) contenuto in $T_r(d)$, così come gli archi (r, a), (a, f) e (f, g) siano ancora parte della soluzione ottima.

Prima di inizializzare la procedura, è indispensabile calcolare i costi ridotti per tutti gli archi. Infatti saranno proprio queste quantità ad essere utilizzate nella successiva chiamata dell'algoritmo di Dijkstra. Per farlo basta ricordare che $\bar{c}_{ij} = c_{ij} + \pi_i - \pi_j$ per ogni arco (i, j) e che, per ogni nodo $i \in N$, π_i non è altro che la lunghezza del cammino minimo da r a i, cioè l'etichetta stessa restituita dall'algoritmo di Dijkstra. Si noti in particolare che i costi ridotti sono tutti positivi (ammissibilità duale) e sono pari a 0 per gli archi che compongono la soluzione corrente (Scarti Complementari). In Figura 3.3 rappresentiamo tali costi ridotti posti in corrispondenza dei rispettivi archi del grafo.
Ora avviamo la procedura di riottimizzazione inizializzando i valori $d(i)$ come sopra descritto: poniamo all'iterazione 0 $d(j) = 0$ per ogni $j \in N$, $j \neq k$, mentre sia $d(k) = \bar{c}'_{gk} =$

-5. Impostiamo g come predecessore di k, mentre per tutti gli altri nodi manteniamo come predecessori i vertici che li precedono nell'albero dei cammini minimi attualmente ottimo.

Osservazione 3. *Inizializzare $d(k)$ con il valore -5 e porre $p(k) = g$ ha un significato preciso che può essere descritto come segue. Se dopo il cambio di costo decidiamo di far entrare (g, k) nell'albero dei cammini minimi, cioè $p_k = p_g \cup (g, k)$ (dove come al solito p_i indica un cammino minimo da r a i), allora con questa scelta riduciamo il costo del cammino minimo p_k esattamente di 5. Le iterazioni successive hanno lo scopo di propagare questa "contrazione di costo" lungo i successori di k all'interno del grafo. Fintanto che incontriamo dei valori di d negativi, troviamo dei cammini minimi con costo minore rispetto alla soluzione precedente.*

Osservazione 4. *Supponiamo che stiamo scansionando, ad una certa iterazione, il nodo i. Per come abbiamo definito l'algoritmo di Dijkstra, non tutti i nodi di $FS(i)$ diverranno candidati dopo questa operazione, bensì solo quelli per cui effettivamente aggiorniamo il valore d (che sarà nel nostro caso negativo). Ciò comporta che l'algoritmo non deve necessariamente eseguire $n = 13$ iterazioni, cioè non è richiesto che esamini tutti i nodi del grafo, bensì giunge a terminazione una volta che l'insieme dei nodi candidati Q risulta vuoto.*

Nel nostro caso, la procedura di riottimizzazione richiede 4 iterazioni, come mostrato in Tabella 3.2. Una volta giunto a termine, l'algoritmo di Dijkstra restituisce un indice di quanto il cammino minimo si sia contratto dopo il cambio di costo. La nuova soluzione duale si ottiene aggiornando π_i per ogni i, cioè sommando a π_i i valori restituiti da questa chiamata:

$$\pi_i \mapsto \pi_i + d(i)$$

I nodi i per i quali il cammino minimo p_i si riduce di costo sono c, j, k e l. Ad esempio per il vertice c il precedente cammino minimo aveva costo $\pi_c = 10$, mentre dopo la riottimizzazione esso si riduce a $\pi_c + d(c) = 10 - 1 = 9$. Similmente, per il nodo j aggiorniamo il valore $\pi_j = 12$ in $\pi_j + d(j) = 7$ e così via. Infine, attraverso la lista dei predecessori si può ricostruire la nuova versione dell'albero dei cammini minimi. In Figura 3.4 lo si vede rappresentato.

Iter	r	a	b	c	d	e	f	g	h	i	j	k	l
0	$0,-$	$0,r$	$0,r$	$0,i$	$0,r$	$0,f$	$0,a$	$0,f$	$0,b$	$0,b$	$0,k$	$-5,g$	$0,i$
1	$0,-$	$0,r$	$0,r$	$0,i$	$0,r$	$0,f$	$0,a$	$0,f$	$0,b$	$0,b$	$-5,k$	$-5,g$	$-4,k$
2	$0,-$	$0,r$	$0,r$	$-1,j$	$0,r$	$0,f$	$0,a$	$0,f$	$0,b$	$0,b$	$-5,k$		$-4,k$
3	$0,-$	$0,r$	$0,r$	$-1,j$	$0,r$	$0,f$	$0,a$	$0,f$	$0,b$	$0,b$			$-4,k$
4	$0,-$	$0,r$	$0,r$	$-1,j$	$0,r$	$0,f$	$0,a$	$0,f$	$0,b$	$0,b$			

Tabella 3.2: Iterazioni dell'algoritmo di Dijkstra per la riottimizzazione. Per ciascun passo è semplice riconoscere i nodi candidati: essi sono i nodi che presentano d negativo.

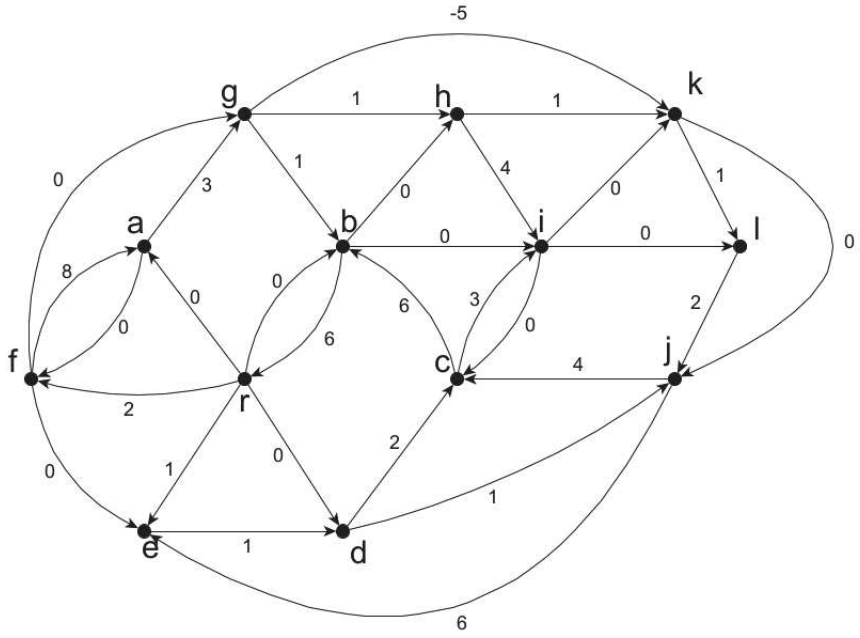

Figura 3.3: Grafo con i costi ridotti modificati in base alla soluzione duale ottima.

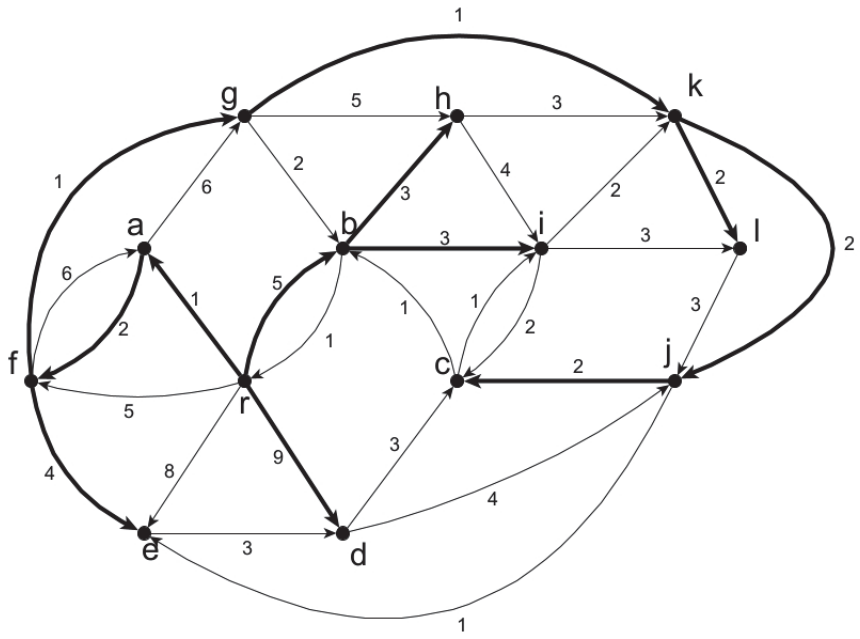

Figura 3.4: La soluzione dopo il cambio di costo.

Costo computazionale

Per quanto questo algoritmo possa richiedere delle attenzioni particolari in fase di inizializzazione, esso resta comunque una procedura *Dijkstra-like*. Infatti la parte sostanziale del calcolo della soluzione viene affidata ad una chiamata dell'algoritmo di Dijkstra. Durante l'esecuzione, poiché abbiamo supposto che i costi degli archi siano non negativi, ogni nodo verrà inserito in Q al massimo una volta, cioè, per ogni nodo scansionato i, l'algoritmo imposta il valore $d(i)$ al suo valore ottimo in modo definitivo. Usando la terminologia già introdotta, diremo che l'algoritmo è *permanente*.
Se decidiamo di implementare la procedura di Dijkstra tramite le due tecniche descritte nel Capitolo 2, avremo un costo computazionale di $O(m + n \log n)$ usando i Fibonacci heaps, oppure un costo di $O(m + C)$, dove $C = -\bar{c}'_{uv}$ tramite l'implementazione di Dial. Dato che $C = -\bar{c}'_{uv} \leq c_{uv} - c'_{uv}$, C di fatto è una misura della perturbazione del costo del problema in generale. Ne segue che, se succede che il costo di (u, v) cambia "poco", allora tale implementazione è particolarmente conveniente. Se ad esempio C è costante o più in generale C è $O(m)$, allora la struttura dati a one-level bucket permette di implementare l'algoritmo di riottimizzazione addirittura in tempo lineare. In conclusione:

Proposizione 3.2.2. *La procedura presentata per riottimizzare cammini minimi in cui il costo di un solo arco (u, v) cambia esegue la riottimizzazione in $O(m + \min\{n \log n, C\})$, dove $C = -\bar{c}'_{uv}$.*

Concludiamo la sezione con un'ultima considerazione che riguarda la presenza di più archi soggetti ad una perturbazione di costo. Anche se ipotizziamo che G non contenga cicli di costo negativo dopo il cambio di costi, una riottimizzazione eseguita rispetto a un sottoinsieme di archi di K potrebbe introdurre dei cicli di costo negativo. Questo tuttavia si rivela un problema facilmente risolvibile: per evitare che ciò accada è sufficiente riottimizzare prima rispetto agli archi $(u, v) \in K$ tali che $\bar{c}'_{uv} > 0$ e successivamente rispetto a quelli per cui $\bar{c}'_{uv} < 0$. Questo stesso ordine verrà seguito per approcci di riottimizzazione più generali, come il seguente.

3.3 Algoritmo generale: fase duale e primale

Supponiamo di utilizzare l'approccio precedentemente descritto per riottimizzare una soluzione ottima T_r^*. Se sono cambiati i costi di k archi, cioè $|K| = k$, allora tale procedura deve essere applicata k volte per ottenere il risultato ottimo. Ciò è evidentemente inefficiente per valori di k molto grandi.
Bisogna dunque scegliere una partizione di K differente. Partiamo dalle seguenti osservazioni.

Osservazione 5. *Se l'insieme K è tale che $\bar{c}'_{ij} \geq 0$ per ogni $(i,j) \in K$, allora $\pi^{(r)}$ è ancora un vettore duale ammissibile dopo il cambio di costi degli archi.*

Osservazione 6. *Se l'insieme K è tale che $\bar{c}'_{ij} < 0$ per ogni $(i,j) \in K$, allora potrebbe risultare conveniente decomporre K in sottoinsiemi aventi la proprietà di poter essere riottimizzati attraverso una procedura che sia permanente, cioè un algoritmo che raggiunge l'ottimalità con una sola chiamata (una per ciascun sottoinsieme di K).*

Sulla base di questi suggerimenti, innanzitutto partizioniamo K in due sottoinsiemi, K^+ e K^-, definiti come segue:

$$K^+ = \{(i,j) \in K : \bar{c}'_{ij} \geq 0\}$$
$$K^- = \{(i,j) \in K : \bar{c}'_{ij} < 0\}$$

Se $K^+ \neq \emptyset$, allora l'algoritmo opera una prima fase sul sottoinsieme K^+, cioè tenendo in considerazione solo i cambi di costi per gli archi di K^+. La Osservazione 5 suggerisce di chiamare questo passo **fase duale**. Essa verrà descritta nel dettaglio in seguito.
Se si ha che $K^- = \emptyset$, allora si può affermare che al termine della fase duale si è giunti alle soluzioni ottime per il problema e dunque l'algoritmo termina.
Se tuttavia $K^- \neq \emptyset$ allora K^- verrà prima eventualmente aggiornato in base ai risultati della fase duale e poi decomposto in h sottoinsiemi $K_1^-, \ldots, K_h^-$ (come decomporre K^- in modo efficace verrà spiegato in dettaglio nel seguito). Su ciascuno di essi verrà infine applicata la cosiddetta **fase primale**. Durante la i-esima fase primale, la riottimizzazione verrà eseguita rispetto agli archi di K_i^- attraverso una procedura di Dijkstra permanente, come richiesto nell'Osservazione 6. Vediamo ora nello specifico il funzionamento delle due fasi.

3.3.1 Fase duale

La fase duale riceve in input le soluzioni ottime correnti $T_r^* = (N, A_r)$ e $\pi^{(r)}$, i costi ridotti $\bar{c}_{ij}$ per ogni $(i,j) \in A$ e i nuovi costi c'_{ij} per ogni $(i,j) \in K^+$.
Come prima cosa calcoliamo i costi ridotti modificati $\bar{c}'_{ij}$ per ogni arco $(i,j) \in A$. Particolare importanza hanno gli archi di $T_r^* = (N, A_r)$. Se $\bar{c}'_{ij} = 0$ per ogni $(i,j) \in A_r$, allora T_r^* e $\pi^{(r)}$ rimangono ancora le soluzioni ottime dopo il cambio di costo. Infatti con questa condizione $\pi^{(r)}$ soddisfa ancora ad uguaglianza i corrispondenti vincoli duali e dunque per il Teorema degli Scarti Complementari T_r^* è soluzione ottima. Si osservi che presuporre $\bar{c}'_{ij} = 0 \ \forall (i,j) \in A_r$ corrisponde alla situazione in cui tutti gli archi che aumentano di

costo non appartengono alla soluzione ottima corrente ed è ragionevole che in tal caso l'ottimalità venga preservata.
Se invece esiste almeno un arco di A_r per il quale $\bar{c}'_{ij} > 0$, allora è necessario avviare una procedura di riottimizzazione la quale ha lo scopo di ripristinare le condizioni degli Scarti Complementari (cioè i vincoli duali soddisfatti ad uguaglianza dalla nuova soluzione duale). Per una tale riottimizzazione si possono adottare diverse strategie: si può ad esempio applicare un qualunque algoritmo duale (come quelli descritti in [7]), in virtù del fatto che l'ammissibilità duale è preservata dopo che i costi degli archi di K^+ sono stati modificati.

Possiamo anche affidare la fase duale all'algoritmo di Dijkstra, in modo del tutto simile al caso precedentemente trattato in cui l'aumento di costo riguardava un arco solo. Basterà assegnare ad ogni arco del grafo il corrispondente costo ridotto modificato rispetto alla soluzione ottima corrente $\pi^{(r)}$ ed avviare poi una procedura di Dijkstra standard. In questo caso la struttura dati one-level bucket di Dial è ancora una volta una scelta vantaggiosa in fase di implementazione. Infatti se definiamo

$$C_d = \max \left\{ \sum_{(i,j)\in P} \bar{c}'_{ij} \colon P \text{ è un cammino di } T_r^* \right\} \tag{3.5}$$

allora la riottimizzazione rispetto a K^+ ha un costo computazionale pari a $O(m + C_d)$. Se chiamiamo $\bar{c}'_{\max}$ la massima perturbazione di costo rispetto agli archi di K^+, allora vale che

$$C_d \leq \min\{ \sum_{(i,j)\in K^+ \cap A_r} \bar{c}'_{ij},\ n\bar{c}'_{\max}\}$$

In altre parole C_d varia in funzione della perturbazione dei costi rispetto a K^+. Ciò implica che a priori possiamo aspettarci una riottimizzazione tanto più veloce quanto più la perturbazione dei costi è bassa.
Si noti che ancora una volta la procedura di Dijkstra risulta permanente poiché la riottimizzazione rispetto a K^+ avviene con una singola chiamata dell'algoritmo.
Possiamo infine affermare che questa strategia risolutiva funziona correttamente: ciò è una diretta conseguenza della correttezza dell'algoritmo di Dijkstra (Teorema 2.2.1) e della Proposizione 3.2.1.

Al termine della fase duale, se $K^- \neq \emptyset$, allora sarà necessario eseguire una o più fasi primali per giungere alla soluzione finale. Bisogna però tenere in considerazione che, come conseguenza della fase duale, la soluzione duale ottima è cambiata. In particolare, se chiamiamo π^* la nuova soluzione duale, si ha che $\pi_i^* \geq \pi_i^{(r)}$ per ogni $i \in N$: infatti, per effetto di un aumento di costi, il costo dei cammini minimi aumenta o al più resta invariato.
Per passare alle fasi successive, bisogna prima aggiornare i costi ridotti modificati rispetto alla soluzione π^* e per farlo ridefiniamo i costi ridotti modificati nel seguente modo:

$$\bar{c}'_{ij} := c'_{ij} + \pi_i^* - \pi_j^*$$

È dunque possibile che alcuni archi che precedentemente avevamo posto in K^- presentino ora un costo non negativo. Si noti però che ciò è possibile solo per archi non appartenenti all'albero dei cammini minimi restituito dalla fase duale. Per quanto detto in precedenza, se un arco (i, j) non appartenente alla soluzione ottima corrente è soggetto ad un aumento di costo (cioè $\bar{c}'_{ij} \geq 0$), allora la soluzione ottima non cambia. Come conseguenza di tutte queste considerazioni, possiamo definire la seguente regola operativa: se al termine della fase duale, ricalcolando i costi ridotti modificati rispetto a π^*, otteniamo che alcuni archi di K^- ora hanno costo ridotto modificato non più negativo, allora possiamo semplicemente rimuovere tali archi da K^-, aggiornare i relativi costi ridotti nel grafo e procedere con le fasi successive.
Ciò significa che, conclusa la fase duale, possiamo automaticamente escludere degli archi da K^- semplicemente guardando i corrispondenti costi ridotti modificati, ottenendo così un ulteriore margine di miglioramento in termini di velocità dell'algoritmo.

3.3.2 Fase primale

L'idea che sta alla base delle fasi primali è di sfruttare la struttura del grafo indotto dagli archi $(i, j) \in A$ tali che $\bar{c}'_{ij} \leq 0$ per decomporre K^- in h sottoinsiemi $K_1^-, \ldots, K_h^-$.
Sarà utile per il ragionamento considerare il seguente sottografo di G. Sia $G_0^- = (N, A_0^-)$ il grafo formato da tutti gli archi il cui costo ridotto è non positivo. In altre parole, $A_0^- = \{(i, j) \in A \colon \bar{c}'_{ij} \leq 0\}$. Si osservi che G_0^-, per come è definito, contiene sia gli archi di K^- che gli archi con costo ridotto zero. In particolare, esso contiene gli archi della soluzione ottima corrente, cioè quella restituita dalla fase precedente appena conclusa. Il motivo per cui conviene considerare questi archi verrà chiarito in seguito, ma comunque è legato alla possibilità di dividere K in un numero "piccolo" di sottoinsiemi.
Si osservi poi che possiamo sempre costruire G_0^- in modo tale che sia aciclico. Giustifichiamo questa affermazione: se G_0^- contiene un ciclo, allora tale ciclo ha costo pari a 0, poiché, per come abbiamo costruito l'algoritmo, nessuna fase precedentemente eseguita genera cicli di costo (ridotto) negativo. In questa situazione, è possibile eliminare tutti i cicli a costo zero del grafo: è sufficiente rimuovere da G_0^- un arco per ciascun ciclo. Senza perdita di generalità possiamo assumere che gli archi rimossi da G_0^- per evitare questi cicli a costo zero non appartengano alla soluzione corrente, cioè all'albero di cammini minimi attualmente ottimo. Alla luce di ciò, possiamo concludere che G_0^- è un albero ricoprente con radice in r, più alcuni archi aggiuntivi.

Il punto cruciale della fase primale consiste nello stabilire come dividere K^- nei sottoinsiemi $K_1^-, \ldots, K_h^-$. Per ogni $i = 1, \ldots, h$ K_i^- genera un sottografo di G, chiamiamolo $G(K_i^-)$, il quale è contenuto in G_0^-. Fondamentale è la seguente definizione.

Definizione 3.3.1. Un **sottografo Dijkstra-permanente** è un sottografo di G_0^- il cui insieme di archi può essere riottimizzato attraverso una singola chiamata della procedura permanente di Dijkstra, indipendentemente dai costi (non negativi) degli archi che non appartengono al sottografo.

Osservazione 7. *Questa definizione è di fondamentale importanza. Infatti se vogliamo sfruttare l'algoritmo di Dijkstra per la riottimizzazione rispetto a K^-, dobbiamo essere*

certi che questo funzioni correttamente. Essendoci molteplici archi di G che presentano costo ridotto negativo, la correttezza di tale algoritmo non è certa. Se però possiamo stabilire a priori che un grafo è Dijkstra-permanente, allora non solo possiamo usare l'algoritmo di Dijkstra per risolvere il problema dato, ma anche tali considerazioni sono sufficienti per dedurre la correttezza della fase primale.

Cerchiamo dei sottoinsiemi K_i^- che godano di questa proprietà. In altre parole, $G(K_i^-) \subseteq G_0^-$ dovrà essere scelto in modo tale che la sua struttura permetta la riottimizzazione dei suoi stessi archi con una sola chiamata della procedura permanente di Dijkstra. Segue una descrizione schematica della fase primale.

Procedura di riottimizzazione: fase primale

1. Si costruisca un sottografo Dijkstra-permanente di G_0^- che contiene almeno un arco di K^-. Sia esso $G(K_i^-) = (N(K_i^-), A(K_i^-))$.
   Sia $K_i^- = A(K_i^-) \cap K^-$ (in tal modo K_i^- è ben definito, poiché $A(K_i^-)$ potrebbe contenere archi con costo ridotto zero).

2. Si cambi il costo degli archi in K_i^-, ignorando il cambio di costo per tutti gli altri archi in K^- (dunque $\bar{c}'_{ij} = \bar{c}_{ij}$ per ogni $(i,j) \in K^- \setminus K_i^-$).

3. Si inizializzi, per ogni nodo $i \in N$, $d(i) = 0$.

4. Si visiti il grafo aciclico $G(K_i^-)$ in ordine topologico e ad ogni nodo v esaminato si assegni il corrispondente valore permanente $d(v)$ come segue:

   $$d(v) \coloneqq \min\{d(u) + \bar{c}'_{uv} \colon (u,v) \in BS(v) \cap A(K_i^-)\}$$

   Contestualmente, si aggiorni la lista dei predecessori $p = [p_i]$ di conseguenza.
   Per completare l'inizializzazione dell'algoritmo, si crei l'insieme dei nodi candidati Q inserendo tutti i nodi v che presentano $d(v)$ negativo:

   $$Q \coloneqq \{v \in N(K_i^-) \colon d(v) < 0\}$$

5. Si chiami ora una procedura di Dijkstra sull'insieme Q precedentemente inizializzato. Per ipotesi tale procedura è permanente. L'obiettivo è quello di "propagare" i valori negativi $d(v)$ a tutti i nodi del grafo, alla ricerca di nuovi cammini di costo minimo. Giunti al termine dell'algoritmo si aggiorni la soluzione duale corrente, che chiamiamo $\pi^{(r)}$, come già visto precedentemente:

   $$\pi_i^{(r)} \coloneqq \pi_i^{(r)} + d(i) \quad \forall i \in N$$

6. Si aggiornino i costi ridotti $\bar{c}'_{ij}$ in base alla soluzione trovata. Si aggiorni infine l'insieme K^-, rimuovendo tutti gli archi che hanno costo ridotto non negativo e si modifichi G_0^- di conseguenza.

Si itera questo procedimento su tutti i sottoinsiemi $K_1^-, \ldots, K_h^-$ fino a che K^- risulta vuoto. A tal punto l'algoritmo termina.

Proposizione 3.3.1. *Al termine di ciascuna fase primale nessun arco $(i, j) \notin K^-$ ha costo ridotto negativo. Inoltre, la cardinalità di K^- decresce strettamente.*

Da ciò si può dedurre che:

Proposizione 3.3.2. *L'algoritmo di riottimizzazione esegue non più di $|K^-|$ fasi primali.*

3.3.3 Come trovare grafi Dijkstra-permanenti

Una questione in sospeso riguarda i grafi Dijkstra-permanenti. Determinare se un sottografo di G_0^- gode di questa proprietà è fondamentale per l'esecuzione delle fasi primali, ma non è ancora chiaro come stabilirlo a priori. Siamo dunque interessati a trovare delle condizioni sufficienti (che in qualche modo devono riguardare la struttura del sottografo) per concludere che un certo sottografo $G(K_i^-)$ sia Dijkstra-permanente. Analizziamo il problema con degli esempi.

Esempio 3.3.1. Supponiamo che un certo grafo contenga al suo interno il sottografo $G(K_i^-) = (N(K_i^-), A(K_i^-))$ rappresentato in Figura 3.5. Gli interi associati agli archi rappresentano i costi ridotti modificati $\bar{c}'_{ij} = c'_{ij} + \pi_i - \pi_j$ rispetto alla soluzione corrente. Ci chiediamo se questo sottografo è Dijkstra-permanente: in altre parole, è vero che una singola chiamata dell'algoritmo di Dijkstra (inizializzato come sopra descritto per la fase primale) dà come risultato i corretti valori dei costi dei cammini minimi $d(v)$ per i nodi $v \in N(K_i^-)$? Si noti che nessuna ipotesi è stata fatta sugli altri archi del grafo, se non che essi non possono avere costo ridotto negativo.
Visitiamo $G(K_i^-)$ in ordine topologico ed inizializziamo d come indicato al punto (4) della fase primale; otteniamo:

$$d(a, b, c, d, e, f, g) = (0, 0, -4, -4, -5, -7, -10)$$

Inseriamo in Q i nodi aventi valore negativo, cioè $Q = \{c, d, e, f, g\}$, e chiamiamo una procedura di Dijkstra sull'insieme di nodi candidati così inizializzato.
Alla prima iterazione verrà scansionato il nodo g, assegnando in modo definitivo il valore $d(g) = -10$. Tale assegnazione è corretta, poiché $d(g)$ non potrebbe essere minore: ciò si deduce dal fatto che tutti gli archi al di fuori di $G(K_i^-)$ hanno costo ridotto non negativo. Nelle iterazioni successive, a tutti i nodi i che vengono scansionati viene associato un valore $d(i)^*$ permanente negativo (non minore di -10) che è il minimo possibile: infatti se così non fosse, allora dovrebbe esistere un cammino da r a i non passante per alcun vertice di K_i^- con costo minore di $d(i)^*$, ma ancora una volta ciò è impossibile vista la non negatività dei costi degli archi al di fuori di K_i^-.
Una chiamata dell'algoritmo di Dijkstra restituisce i valori corretti di $d(i)$ per ogni nodo $i \in G(K_i^-)$: dunque, questo è un esempio di un sottografo Dijkstra-permanente, corrispondente all'insieme $K_i^- = \{(a, c), (b, c), (d, e), (e, f), (e, g)\}$. Si osservi che l'arco (c, d) con costo ridotto 0 funge da collegamento tra i due sottografi indotti dagli archi $\{(a, c), (b, c)\}$ e $\{(d, e), (e, f), (e, g)\}$ appartenenti a K^-. Questo permette di trattare contemporaneamente i due sottoinsiemi di archi.

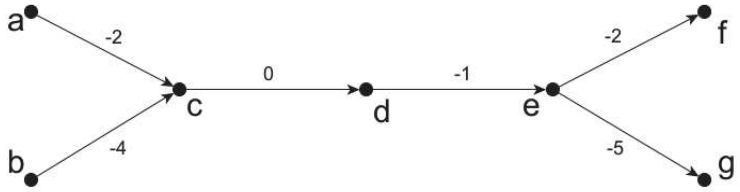

Figura 3.5: Un sottografo Dijkstra-permanente.

Esempio 3.3.2. Si consideri il sottografo $G(K_i^-) = (N(K_i^-), A(K_i^-))$ rappresentato in Figura 3.6. Visitando i nodi in ordine topologico e applicando la regola (4), si ottiene $d(a, b, c, d) = (0, -3, -1, -5)$ e come conseguenze $Q = \{b, c, d\}$. È chiaro che durante la prima iterazione dell'algoritmo di Dijkstra viene scansionato il nodo d assegnando in modo permanente il valore $d(d) = -5$.
Assumiamo ora che, al di fuori di $G(K_i^-)$ esista un arco (b, c) con costo $\bar{c}'_{bc} = 1$. In questa situazione, seguendo il cammino $\{(a, b), (b, c), (c, d)\}$, troveremmo un cammino con costo pari a -6, minore di quello assegnato dall'algoritmo. Ne segue che per ottenere la corretta riottimizzazione non è sufficiente una chiamata della procedura di Dijkstra (ne sono necessarie due): per definizione, questo sottografo non è Dijkstra-permanente.

Esempio 3.3.3. In Figura 3.7 si mostra un altro caso in cui il sottografo non è Dijkstra-permanente. In questo esempio, la visita topologica restituisce $d(a, b, c, d) = (0, -3, 0, -4)$ e $Q = \{2, 4\}$. Nuovamente, la presenza di un arco (b, c) con costo ridotto $\bar{c}'_{bc} = 1$ causerebbe il reinserimento del nodo d in Q con un'altra chiamata dell'algoritmo per ottenere il valore effettivo $d(d) = -5$, rendendo la procedura non permanente.

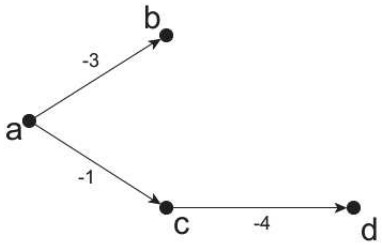

Figura 3.6: Un sottografo che non è Dijkstra-permanente.

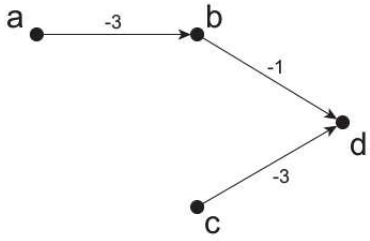

Figura 3.7: Un altro sottografo non Dijkstra-permanente.

Dagli esempi sopra descritti si deduce che, in generale, se un grafo $G(K_i^-)$ contiene sottografi isomorfi a quelli descritti negli Esempi 3.3.2 e 3.3.3, allora c'è la possibilità che la procedura di Dijkstra non sia permanente. Ciò suggerisce la seguente definizione.

Definizione 3.3.2. Un **cammino stellato centrato in i e j** (*Starpath*) è un sottografo di $G_0^- = (N, A_0^-)$ costituito da un cammino orientato da i a j con l'aggiunta di due insiemi di archi:

$$\overline{BS}(i) = BS(i) \cap K^- \qquad \overline{FS}(j) = FS(j) \cap K^-$$

Il sottografo rappresentato in Figura 3.5 è un cammino stellato centrato in c ed e. Si noti che la definizione di cammino stellato include anche il caso in cui $i = j$.
Si consideri un cammino stellato centrato in i e j. Sia (u, i) l'arco di $\overline{BS}(i)$ con costo ridotto minimo ($v = i$ nel caso in cui $\overline{BS}(i) = \emptyset$):

$$\bar{c}'_{ui} = \min\{\bar{c}'_{wi} \colon (w, i) \in \overline{BS}(i)\} \tag{3.6}$$

Si chiami poi (j, v) l'arco uscente da j con costo ridotto minimo, assumendo che esso non sia vuoto (altrimenti si ponga $v = j$):

$$\bar{c}'_{jv} = \min\{\bar{c}'_{jw} \colon (j, w) \in \overline{FS}(i)\} \tag{3.7}$$

Lemma 3.3.1. *Sia S un cammino stellato centrato in i e j e sia $P_{uv} \subseteq S$, cammino di estremi u e v definiti come sopra. Allora P_{uv} appartiene all'albero di cammini minimi ottenuto riottimizzando rispetto agli archi del cammino stellato.*

Dimostrazione. Ricordiamo che abbiamo assunto che nessun ciclo di costo negativo viene creato in G dopo il cambio di costi. Necessariamente il cammino minimo riottimizzato P_{ri} da r a i include l'arco (u, i), poiché esso, per definizione, è l'arco che permette di ottenere il costo minimo tra tutti gli archi di $BS(i)$. Il cammino minimo riottimizzato da r a v si ottiene concatenando P_{ri} al cammino da i a v contenuto nel cammino stellato, poiché ancora una volta gli archi con costo ridotto negativo sono più vantaggiosi rispetto agli altri archi del grafo. Segue dunque che P_{uv} fa parte dell'albero dei cammini minimi. □

Proposizione 3.3.3. *Ogni cammino stellato è un sottografo Dijkstra-permanente.*

Dimostrazione. Sia S un cammino stellato contenuto in un grafo G. Durante una chiamata dell'algoritmo di Dijkstra, un certo numero di nodi di G verrà prima contrassegnato come candidato e ad ogni iterazione ciascun nodo, uno ad uno, verrà scansionato e il corrispondente valore d verrà assegnato in modo permanente (restituendo d^*).
La procedura di Dijkstra non funziona correttamente se e solo se esiste un cammino, diverso da quello calcolato dall'algoritmo, tale da avere costo minore di d^*. Il Lemma 3.3.1, unitamente al fatto che nessun arco al di fuori di S ha costo ridotto negativo, assicura che ciò è impossibile. Dunque una procedura di Dijkstra è sufficiente per ottenere la soluzione corretta, da cui si deduce che S è Dijkstra-permanente. □

3.3.4 Esempio

Vediamo come applicare tutti i concetti finora formulati riguardo la teoria della riottimizzazione dei cammini minimi in un esempio. Consideriamo nuovamente il grafo $G = (N, A)$ rappresentato in Figura 3.1, i corrispondenti costi e il problema dell'albero dei cammini minimi con radice r. Conosciamo già le corrispondenti soluzioni ottime, cioè l'albero ricoprente raffigurato in Figura 3.2 e il vettore duale $\pi = [\pi_i] \in \mathbb{Z}^N$ le cui componenti hanno i seguenti valori:

v	r	a	b	c	d	e	f	g	h	i	j	k	l
π_v	0	1	5	10	9	7	3	4	8	8	12	10	11

Ricordiamo che per ogni $v \in N$ il valore π_v rappresenta il costo del cammino minimo da r a v.
Nel contesto della riottimizzazione ci poniamo nell'ambito più generale, quello che abbiamo studiato nel dettaglio fino a questo momento, cioè quello in cui un qualsiasi sottoinsieme di archi di A subisce una perturbazione di costo, sia in aumento che in diminuzione. Supponiamo allora che alcuni archi cambino costo, ad esempio

$$\begin{aligned} c'_{af} &= 4 & \qquad c'_{bi} &= 8 \\ c'_{ed} &= 0 & \qquad c'_{hk} &= 4 \\ c'_{il} &= 2 & \qquad c'_{kj} &= 1 \\ c'_{rb} &= 7 & \qquad c'_{rd} &= 2 \end{aligned}$$

e applichiamo la tecnica generale di riottimizzazione descritta nella Sezione 3.3.

L'insieme K composto dagli archi con un nuovo costo è

$$K = \{(a, f), (b, i), (e, d), (h, k), (i, l), (k, j), (r, b), (r, d)\}$$

Per prima cosa calcoliamo $\bar{c}'_{ij} = c'_{ij} + \pi_i - \pi_j$ per tutti gli archi del grafo, ricordando che $c'_{ij} = c_{ij}$ per ogni archi $(i, j) \notin K$. Abbiamo in particolare che

$$\begin{aligned} \bar{c}'_{af} &= 2 & \bar{c}'_{bi} &= 5 & \bar{c}'_{hk} &= 2 & \bar{c}'_{rb} &= 2 \\ \bar{c}'_{ed} &= -2 & \bar{c}'_{il} &= -1 & \bar{c}'_{kj} &= -1 & \bar{c}'_{rd} &= -7 \end{aligned}$$

dunque

$$\begin{aligned} K^+ &= \{(a, f), (b, i), (h, k), (r, b)\} \\ K^- &= \{(e, d), (i, l), (k, j), (r, d)\} \end{aligned}$$

Comincia ora la fase duale.

Fase duale

Cambiamo di costo solo gli archi con costo ridotto $\bar{c}'$ non negativo, cioè gli archi di K^+. Ignoriamo le altre variazioni di costo. Questa fase ha lo scopo di riottimizzare la soluzione rispetto agli archi di K^+, ripristinando le condizioni di ottimalità definite dagli Scarti Complementari. La fase duale è affidata ad una chiamata della procedura di Dijkstra, in cui però agli archi associamo i costi ridotti modificati $\bar{c}'_{ij}$ per ogni $(i, j) \in A$, come mostrato in Figura 3.8. In Tabella 3.3 sono riportate le iterazioni dell'algoritmo di Dijkstra: i valori $d(v)$ associati a ciascun nodo v indicano di quanto il cammino minimo da r a v si sia esteso per effetto dell'aumento dei costi. Si osservi che, come atteso, la procedura non restituisce valori di d maggiori del massimo dei costi ridotti, cioè $\bar{c}'_{bi} = 5$.

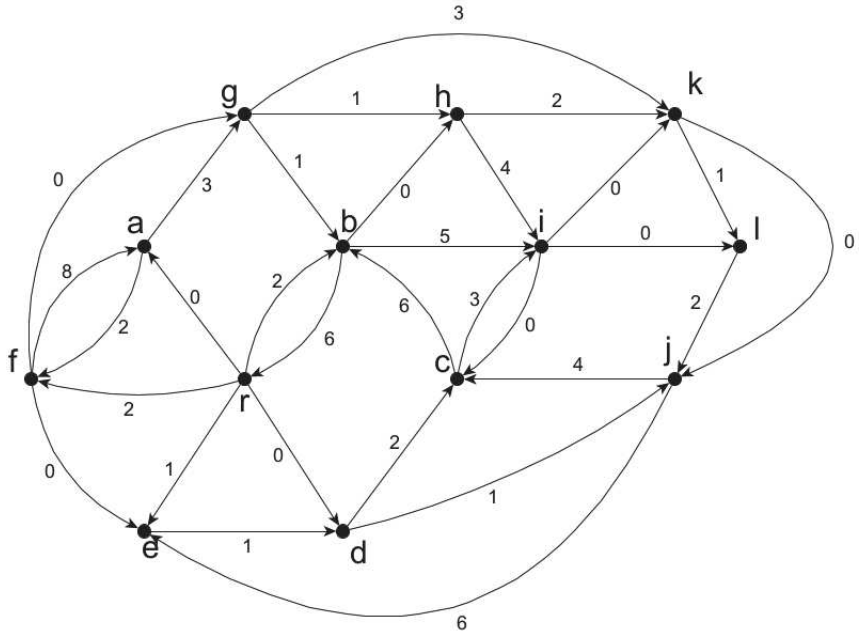

Figura 3.8: Grafo con costi ridotti $\bar{c}'$ rispetto al cambio di costo per K^+.

Una volta che l'algoritmo giunge a termine, aggiorniamo il vettore duale π sommando, per ciascun nodo $v \in N$, π_v alla distanza effettiva $d(v)^*$ restituita dalla procedura e leggibile nella tabella qui sopra. Otteniamo la soluzione aggiornata π^*:

v	r	a	b	c	d	e	f	g	h	i	j	k	l
π^*_v	0	1	7	12	9	8	5	6	10	13	13	14	16

In Figura 3.9 mostriamo la soluzione dopo la riottimizzazione parziale ottenuta grazie alla fase duale. È stato possibile costruirla grazie alla lista di predecessori, che abbiamo man mano aggiornato in parallelo ad ogni iterazione dell'algoritmo.

Iter	r	a	b	c	d	e	f	g	h	i	j	k	l
0	$0,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$
1	$0,-$	$0,r$	$2,r$	$\infty,-$	$0,r$	$1,r$	$2,r$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$
2		$0,r$	$2,r$	$\infty,-$	$0,r$	$1,r$	$2,r$	$3,a$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$	$\infty,-$
3			$2,r$	$2,d$	$0,r$	$1,r$	$2,r$	$3,a$	$\infty,-$	$\infty,-$	$1,d$	$\infty,-$	$\infty,-$
4			$2,r$	$2,d$		$1,r$	$2,r$	$3,a$	$\infty,-$	$\infty,-$	$1,d$	$\infty,-$	$\infty,-$
5			$2,r$	$2,d$			$2,r$	$3,a$	$\infty,-$	$\infty,-$	$1,d$	$\infty,-$	$\infty,-$
6			$2,r$	$2,d$			$2,r$	$3,a$	$2,b$	$7,b$		$\infty,-$	$\infty,-$
7				$2,d$			$2,r$	$3,a$	$2,b$	$5,c$		$\infty,-$	$\infty,-$
8							$2,r$	$2,f$	$2,b$	$5,c$		$\infty,-$	$\infty,-$
9								$2,f$	$2,b$	$5,c$		$5,g$	$\infty,-$
10									$2,b$	$5,c$		$4,h$	$\infty,-$
11										$5,c$		$4,h$	$5,k$
12										$5,c$			$5,k$
13													$5,k$

Tabella 3.3: Iterazioni dell'algoritmo di Dijkstra per la fase duale. Ad ogni passo i nodi candidati sono i nodi che presentano d finito.

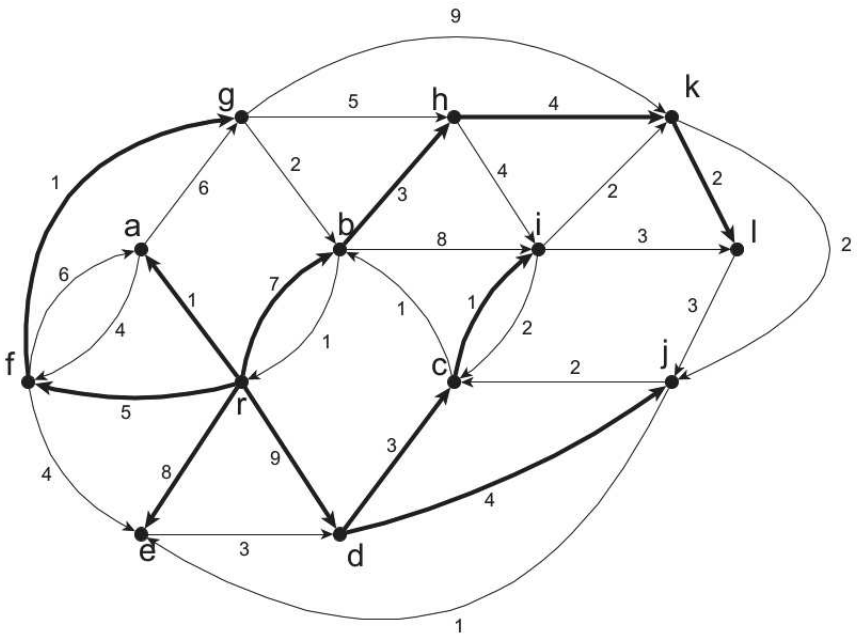

Figura 3.9: Albero dei cammini minimi aggiornato in seguito alla fase duale.

Fase primale

Procediamo ora con le fasi primali. L'obiettivo è riottimizzare la soluzione corrente ottenuta dalla fase duale rispetto ai nuovi costi degli archi di K^-, cioè gli archi con costo ridotto modificato negativo. Calcoliamo come di consueto tali costi ridotti: per ogni $(i,j) \in A$, $\bar{c}'_{ij} = c'_{ij} + \pi^*_i - \pi^*_j$ tenendo conto che $c'_{ij} = c_{ij}$ per tutti gli archi $(i,j) \notin K^-$.

Tutti i costi ridotti sono riportati in Figura 3.11, mentre per gli archi di K^- abbiamo:

$$\bar{c}'_{ed} = -1 \qquad \bar{c}'_{il} = -1 \qquad \bar{c}'_{kj} = 2 \qquad \bar{c}'_{rd} = -7$$

Notiamo che per l'arco (k, j) il costo $\bar{c}'_{kj}$ calcolato rispetto alla soluzione duale è positivo. È successo infatti che per effetto della fase duale la configurazione della soluzione ottima è cambiata tanto da rendere l'arco (k, j) meno "conveniente" rispetto a prima. Infatti il costo ridotto $\bar{c}_{kj}$ aumenta se calcolato rispetto a π^*, così come il costo ridotto modificato $\bar{c}'_{ij}$. Essendo l'arco (k, j) al di fuori dell'albero dei cammini minimi, il fatto che $\bar{c}'_{kj}$ sia positivo non modifica la soluzione ottima. Dunque possiamo semplicemente rimuovere (k, j) da K^- e procedere con le fasi primali.

Sia $G_0^- = (N, A_0^-) \subseteq G$ il grafo contenente gli archi (i, j) tali che $\bar{c}_{ij} \leq 0$. Per procedere con la riottimizzazione rispetto a $K^- = \{(r, d), (e, d), (i, l)\}$, è necessario trovare dei sottografi di G_0^- Dijkstra-permanenti.
Consideriamo il sottografo di G_0^- rappresentato in Figura 3.10, che chiamiamo $G(K^-)$. Si noti che gli archi di $G(K^-)$ sono dati da $K^- \cup \{(d, c), (c, i)\}$. In base alle definizioni date in precedenza, questo sottografo è un *cammino stellato* centrato in d e i e dunque, per la Proposizione 3.3.3, possiamo concludere che si tratta di un grafo Dijkstra-permanente. Abbiamo dunque trovato un grafo Dijkstra-permanente contenente tutti gli archi di K^-: ciò significa che è sufficiente una singola fase primale per completare la riottimizzazione.

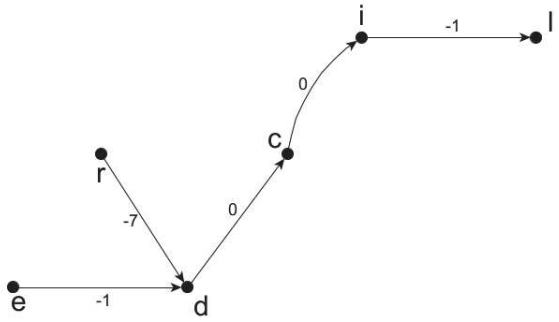

Figura 3.10: Un sottografo di G Dijkstra-permanente.

Inizializziamo dunque una nuova chiamata dell'algoritmo di Dijkstra seguendo le regole per la fase primale. Poniamo prima $d(i) = 0$ per ogni $i \in N$ e poi visitiamo il grafo $G(K^-)$ in ordine topologico aggiornando per ciascun nodo $v \in G(K^-)$ il valore

$$d(v) \coloneqq \min\{d(u) + \bar{c}'_{uv} \colon (u, v) \in BS(v) \cap K^-\}$$

Otteniamo le seguenti assegnazioni:

$$d(r, e, d, c, i, l) = (0, 0, -7, -7, -7, -8)$$

Al passo 0 l'insieme dei nodi candidati è $Q = \{d, c, i, l\}$. Avviamo ora la procedura di Dijkstra per la riottimizzazione finale. La Tabella 3.4 riporta le iterazioni dell'algoritmo.

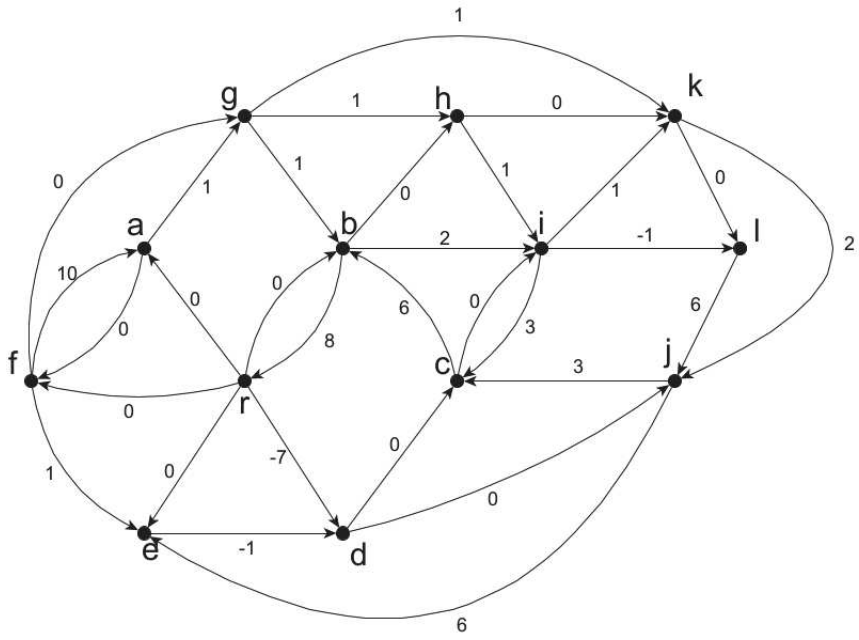

Figura 3.11: Grafo con costi ridotti rispetto alla soluzione duale π^*.

Iter	r	a	b	c	d	e	f	g	h	i	j	k	l
0	$0,-$	$0,r$	$0,r$	$-7,d$	$-7,r$	$0,r$	$0,r$	$0,f$	$0,b$	$-7,c$	$0,d$	$0,h$	$-8,i$
1	$0,-$	$0,r$	$0,r$	$-7,d$	$-7,r$	$0,r$	$0,r$	$0,f$	$0,b$	$-7,c$	$-2,l$	$0,h$	$-8,i$
2	$0,-$	$0,r$	$-1,c$	$-7,d$	$-7,r$	$0,r$	$0,r$	$0,f$	$0,b$	$-7,c$	$-2,l$	$0,h$	
3	$0,-$	$0,r$	$-1,c$		$-7,r$	$0,r$	$0,r$	$0,f$	$0,b$	$-7,c$	$-7,d$	$0,h$	
4	$0,-$	$0,r$	$-1,c$			$0,r$	$0,r$	$0,f$	$0,b$	$-7,c$	$-7,d$	$-6,i$	
5	$0,-$	$0,r$	$-1,c$			$-1,j$	$0,r$	$0,f$	$0,b$		$-7,d$	$-6,i$	
6	$0,-$	$0,r$	$-1,c$			$-1,j$	$0,r$	$0,f$	$0,b$			$-6,i$	
7	$0,-$	$0,r$	$-1,c$			$-1,j$	$0,r$	$0,f$	$-1,b$				
8	$0,-$	$0,r$				$-1,j$	$0,r$	$0,f$	$-1,b$				
9	$0,-$	$0,r$					$0,r$	$0,f$	$-1,b$				

Tabella 3.4: Iterazioni dell'algoritmo di Dijkstra per la fase primale. Ad ogni passo i nodi candidati sono i nodi che presentano d negativo.

Come previsto, i valori di d restituiti dall'algoritmo sono non positivi. In particolare, per ogni vertice $i \in N$, $d(i)^*$ indica di quanto il cammino minimo da r ad i si sia contratto per effetto della riduzione di costi. Possiamo dunque aggiornare nuovamente la soluzione duale sommando a π^* i valori restituiti dalla procedura di Dijkstra. Otteniamo la soluzione finale π, le cui componenti sono:

v	r	a	b	c	d	e	f	g	h	i	j	k	l
π_v	0	1	6	5	2	7	5	6	9	6	6	8	8

Infine, ripercorriamo la lista dei predecessori in Tabella 3.4 così da poter rappresentare l'albero dei cammini minimi ottimo dopo la riottimizzazione (Figura 3.12).

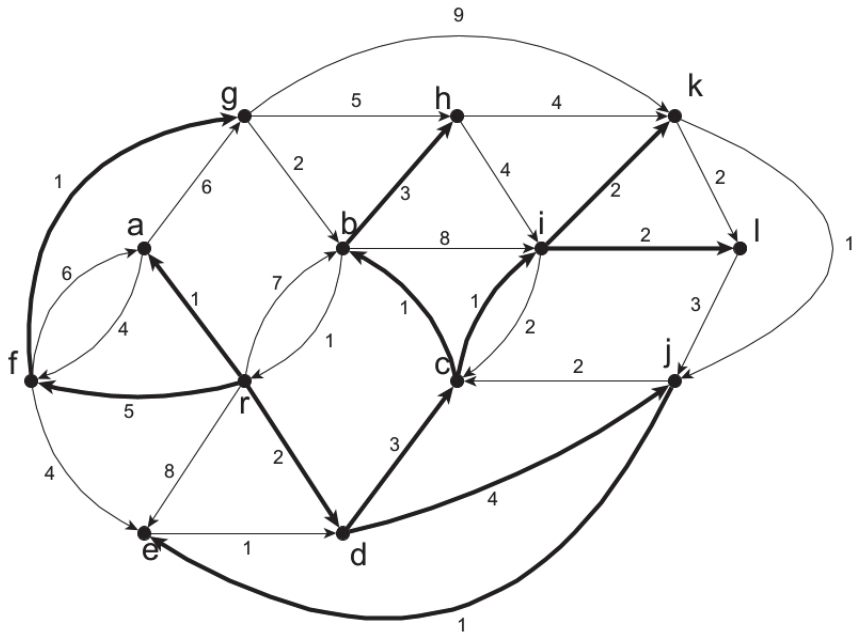

Figura 3.12: Albero dei cammini minimi riottimizzato.

3.3.5 Costo computazionale e conclusioni

Nell'esempio appena considerato, abbiamo trovato il cammino stellato "a occhio", nel senso che prima abbiamo isolato il sottografo $G(K^-)$ e in seguito abbiamo riconosciuto che si tratta di un cammino stellato. In realtà, in un contesto generale, la procedura di identificare i cammini stellati per mezzo dei quali partizionare l'insieme K^- deve essere implementata attraverso un apposito algoritmo, che richiede dunque un ulteriore costo computazionale.

Un metodo è ad esempio il seguente: dati un grafo $G = (N, A)$ con costi ridotti modificati $\bar{c}'_{ij}$ associati ad ogni $(i, j) \in A$ e il sottografo $G_0^- \subseteq G$, si esegue una ricerca in profondità (*depth-first search*) di G_0^- a partire dalla radice r. La ricerca termina quando viene raggiunta una foglia dell'albero dei cammini ottimo e il cammino da r a tale foglia contiene almeno un arco di K^-; chiamiamo (j, v) l'ultimo arco di K^- incontrato percorrendo questo cammino orientato. A questo punto ci si muove a ritroso da j a r seguendo i predecessori ottenuti dalla procedura di ricerca e si determinano due nodi i e u tali che soddisfino (3.7). In questo modo è possibile identificare un cammino stellato centrato in i e j in $O(|A_0^-|) = O(m)$.

Definiamo v come il numero di nodi j di G_0^- aventi almeno un arco di K^- in $BS(j)$:

$$v = |\{j \in V : \text{ esiste } (i, j) \in BS(j) \text{ tale che } (i, j) \in K^-\}|$$

Allora possiamo riformulare la Proposizione 3.3.2 come segue:

Proposizione 3.3.4. *Il numero di fasi primali eseguite dall'algoritmo di riottimizzazione è h, con $h \leq \min\{v, |K^-|\}$.*

A livello computazionale, la componente più costosa di ciascuna fase primale è la chiamata della procedura permanente di Dijkstra. Supponiamo di eseguire una fase primale rispetto ad un sottoinsieme $K_j^- \subseteq K^-$, scelto in modo tale che gli archi di K^- siano contenuti in un cammino stellato. Se decidiamo di implementare tale algoritmo tramite gli *F-heaps* prevediamo un costo computazione non maggiore di $O(m+n\log n)$ per ciascuna fase primale. Se adottiamo l'implementazione di Dial, possiamo usare una struttura dati il cui numero di *buckets* è limitato superiormente da $C(K_j^-) = -\sum_{(i,j)\in K_j^-} \bar{c}'_{ij}$. Ciò significa che questa strategia comporta un costo computazionale pari a $O(m + C(K_j^-))$. Definendo

$$C_p = -\sum_{(i.j)\in K^-} \bar{c}'_{ij} \tag{3.8}$$

che corrisponde ad una misura della perturbazione di costo dovuta al cambio di costi per gli archi di K^-, possiamo enunciare il seguente teorema.

Teorema 3.3.1. *Supponiamo che la procedura di riottimizzazione richieda l'esecuzione di h fasi primali. Allora il costo computazionale complessivo di queste fasi primali è $O(hm + \min\{hn\log n, C_p\})$, dove C_p è definito in (3.8)*

In tale espressione il primo contributo $O(hm)$ è impiegato nella ricerca degli h cammini stellati, mentre il secondo, $O(\min\{hn\log n, C_p\})$, riguarda lo svolgimento delle procedure di Dijkstra.

Si noti che se la perturbazione di costo C_p è $O(mn)$, allora l'approccio risolutivo proposto è non peggiore, per lo meno in teoria, rispetto all'alternativa di applicare un classico algoritmo per la ricerca dei cammini minimi nel caso in cui ci siano degli archi di costo negativo, come ad esempio l'algoritmo di Bellman-Ford. Tale algoritmo infatti calcola l'albero dei cammini minimi in un tempo di $O(mn)$.

In conclusione, in base alle informazioni note riguardo la fase duale e il teorema appena enunciato, possiamo concludere con il seguente risultato.

Teorema 3.3.2. *Il costo computazionale complessivo della tecnica di riottimizzazione proposta è*

$$O(hm + \min\{hn\log n, C_p\} + \min\{n\log n, C_d\})$$

dove C_p è definito in (3.8), C_d è definito in (3.5) e h indica il numero di fasi primali eseguite dall'algoritmo.

In questa discussione abbiamo analizzato un approccio algoritmico che riottimizza cammini minimi nel caso in cui un qualunque sottoinsieme di archi del grafo in questione è soggetto ad un cambio di costo, sia esso maggiore o minore rispetto a prima. Abbiamo analizzato il costo computazionale dell'algoritmo sia in funzione delle dimensioni del grafo (cioè il numero di nodi e di archi), sia in funzione della perturbazione di costo complessiva dovuta alle modifiche apportate al grafo.
Sebbene questo approccio è stato qui presentato a livello puramente metodologico, è ragionevole pensare che esso si dimostri competitivo anche nella pratica. Infatti nei contesti applicativi accade piuttosto frequentemente che la perturbazione dei costi non riguardi

tutto il grafo nella sua complessità, bensì solo piccole e specifiche porzioni di esso. Si consideri ad esempio un'area suburbana soggetta a congestione stradale durante l'ora di punta mattutina. Se si analizza la corrispondente rete di trasporti e si calcolano i cammini minimi aventi origine in quest'area e destinazione il Central Business District (CBD) ogni 15 minuti, allora possiamo osservare che gli archi il cui costo cambia formano un "anello" irregolare centrato nel CBD, il quale si restringe sempre più verso il centro man mano che il tempo passa.
Un altro tipico esempio è la congestione stradale lungo autostrade molto trafficate. La porzione della rete di trasporti i cui archi dimostrano una perturbazione di costi generalmente forma un lungo flusso (l'autostrada stessa insieme alle strade confluenti), facilmente decomponibili in pochi lunghi cammini stellati.

Concludiamo con un ultimo riferimento a [2]. Questo articolo è un resoconto del contributo di P. Festa, S. Fugaro e F. Guerriero le quali, dopo aver implementato l'algoritmo di S. Pallottino e M. G. Scutellà, ne hanno testato l'efficienza in confronto all'approccio risolutivo classico. I test sono stati eseguiti su grafi strutturalmente diversi tra loro e con molteplici perturbazioni di costo. Da questi risultati, non emerge una dominanza netta di una tecnica piuttosto che l'altra, ma in numerose situazioni l'algoritmo di riottimizzazione dimostra una performance migliore rispetto alla risoluzione ex novo. Possiamo quindi considerare il contributo in [2] come una "guida operativa" grazie alla quale, semplicemente osservando la struttura del grafo sotto esame, possiamo scegliere l'approccio risolutivo per il calcolo dei cammini minimi più efficiente.

Bibliografia

[1] Marco Di Summa. Note del corso di ricerca operativa. *Università degli Studi di Padova*, 2012.

[2] Paola Festa, Serena Fugaro, and Francesca Guerriero. Shortest path reoptimization vs resolution from scratch: a computational comparison. *Optimization Methods and Software*, 37(3):1122–1144, 2022.

[3] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[4] Satoru Fujishige. A note on the problem of updating shortest paths. *Networks*, 11(3):317–319, 1981.

[5] Giorgio Gallo. Reoptimization procedures in shortest path problem. *Rivista di Matematica per le Scienze Economiche e Sociali*, 3:3–13, 1980.

[6] Giorgio Gallo and Stefano Pallottino. Shortest path methods: A unifying approach. *Netflow at Pisa*, 1:38–64, 1986.

[7] Stefano Pallottino and Maria Grazia Scutellà. Dual algorithms for the shortest path tree problem. *Networks: An International Journal*, 29(2):125–133, 1997.

[8] Stefano Pallottino and Maria Grazia Scutellà. A new algorithm for reoptimizing shortest paths when the arc costs change. *Operations Research Letters*, 31(2):149–160, 2003.

[9] Laurence A Wolsey and George L Nemhauser. *Integer and Combinatorial Optimization*, volume 55. John Wiley & Sons, 1999.

[10] Xiaoge Zhang, Zili Zhang, Yajuan Zhang, Daijun Wei, and Yong Deng. Route selection for emergency logistics management: A bio-inspired algorithm. *Safety Science*, 54:87–91, 2013.