UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Corso di Laurea in Informatica

ALGORITMI BI-DIREZIONALI PER PROBLEMI DI FLUSSO SU RETE

Relatore: Prof. Giovanni Righini

Tesi di:
Filippo Magi
Matricola: 923142

Anno Accademico 2020-2021

Dedicato a tutti color che, in un modo o nell'altro, mi hanno permesso di essere qui davanti a voi

Prefazione

Il problema del flusso massimo è un problema di ottimizzazione combinatoria, che, data un rete di flusso, mi permette di trovare un flusso ammissibile tale che sia massimo. Il problema, formulato nel 1954 da T.E Harris e F.S. Ross, ha avuto, nel 1956 il primo algoritmo noto, chiamato ad oggi "Algoritmo di Ford-Fulkerson", dal nome dei suoi ideatori, Lester R. Ford e Delbert R. Fulkerson, seppur non implementabile. Negli anni successivi sono state presentate varie implementazioni, tra cui citiamo l'algoritmo di Edmondss-Karp, che aggiunge all'algoritmo di Ford-Fulkerson l'esplorazione attraverso BFS, con complessità computazionale $O(nm^2)$, l'algoritmo di Dinic, l'algoritmo Shortest Augmenting Path, che, esplorando con una DFS, riesce a risolvere il problema in $O(n^2m)$, e nuovi algoritmi per risolvere il problema, come l'algoritmo push-relabel, che, tramite esplorazione FIFO, ha complessità computazionale $O(n^3)$, e, sfruttando gli alberi dinamici, raggiunge complessità computazionale $O(nm \log(n^2/m)$. In questa tesi abbiamo provato a risolvere il problema del flusso massimo tramite una ricerca bidirezionale e ottimizzazione della BFS utilizzata nell'algoritmo di Edmondss-Karp, comparandolo all'algoritmo di Shortest Augmenting Path, anch'esso reso bidirezionale.

Organizzazione della tesi

La tesi è organizzata come segue:

- nel Capitolo 1 presento il problema del flusso massimo e gli algoritmi utilizzati per risolverlo: Ford-Fulkerson, Edmonds-Karp e Shortest Augmenting Path.
- nel Capitolo 2 presento le soluzioni monodirezionali per la ricerca del flusso massimo che sono state utilizzate.
- nel Capitolo 3 presento le strategie usate per l'esplorazione bidirezionale e le soluzioni bidirezionali adottate per la ricerca del flusso massimo.
- nel Capitolo 4 presento la generazione di grafi creati pseudo-casualmente, che verranno usati come benchmark per gli algoritmi presentati.

Indice

Capitolo 1

Problema del flusso massimo

1.1 Rete di flusso

Il problema del flusso massimo è un problema di ottimizzazione combinatoria il cui obiettivo è trasportare simultaneamente da un nodo sorgente Source s ad un nodo destinazione Sink t la maggior quantità possibile di unità di flusso su una rete. La trattazione del problema è presentata nell'ottavo capitolo del libro scritto da Korte [1], che si consiglia per uno studio più approfondito.

Una rete di flusso è una tupla (G, u, s, t), dove G è un digrafo e u è la funzione capacità $u : E(G) \to \mathbb{R}_+$.

Definito ciò, il flusso è una funzione $f : E(G) \to \mathbb{R}_+ | f(e) \leq u(e), \forall e \in E(G)$.

L'eccesso di un flusso f in $v \in V(G)$ è

$$ex_f(v) := \sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e)$$

f soddisfa la legge di conservazione di flusso per un vertice v se $ex_f(v) = 0$.
Data una rete (G, u, s, t), un flusso s-t è un flusso che soddisfa i seguenti requisiti:

- $ex_f(s) \leq 0$
- $ex_f(v) = 0 \, \forall \, v \in V(G) \setminus \{s, t\}$

Il valore del flusso s-t f è $value(f) = -ex_f(s)$.

Il nostro obiettivo quindi è, data una rete (G, u, s, t), trovare un flusso s-t con il valore massimo. Il problema del flusso massimo possiamo scriverlo in programmazione lineare come segue:

$$\max \sum_{e \in \delta^+(s)} x_e - \sum_{e \in \delta^-(s)} x_e$$

$$s.t. \sum_{e \in \delta^+(v)} x_e = \sum_{e \in \delta^-(s)} x_e \ \ (v \in V(G) \setminus \{s,t\})$$
$$x_e \leq u(e) \ \ (e \in E(G))$$
$$x_e \geq 0 \ \ (e \in E(G))$$

.

Da questa PL possiamo dimostrare che il problema del flusso massimo ha sempre una soluzione ottima. Definiamo il taglio s-t in G come l'insieme di archi $\delta^+(X)$, dove $s \in X$ mentre $t \in V(G) \setminus X$. La capacità di un taglio s-t è pari alla somma della capacità dei suoi archi,quindi si dice che un taglio s-t è minimo in (G,u) quando il taglio s-t ha capacità minima in G rispetto a u, è inoltre possibile dimostrare che il flusso massimo può essere al più uguale alla capacità di un taglio s-t minimo.

Definiamo per un grafo G il grafo $\overleftrightarrow{G} := (V(G), E(G) \cup \{\overleftarrow{e} : e \in E(G)\}$, dove per ogni arco $e = (u,v)$, l'arco inverso $\overleftarrow{e} = (v,u)$.

Dato un digrafo G con capacità $u : E(G) \to \mathbb{R}_+$, un flusso f, definiamo capacità residua $u_f : E(\overleftrightarrow{G}) \to \mathbb{R}_+$, con $u :_f (e) = u(e) - f(e)$ e $u_f(\overleftarrow{e}) = f \, \forall e \in E(G)$.

Grazie alla capacità residua possiamo definire il grafo residuale $G_f = (V(G), \{e \in E(\overleftrightarrow{G}) : u_f > 0\})$

Aumentare un flusso f lungo un cammino P in G_f di γ unità significa che $\forall e \in E(P)$, se $e \in E(G)$, allora $f(e)$ aumenta di γ,altrimenti se $e = \overleftarrow{e}_0$ per e_0, allora $f(e_0)$ diminuisce di γ.

Data una rete (G,u,s,t) e un flusso s-t f, un cammino f-aumentante è un cammino s-t nel grafo residuale G_f.

1.2 Algoritmo di Ford-Fulkerson

Dati i concetti appena elencati, possiamo discutere dell'algoritmo di Ford-Fulkerson, datato 1956. Per semplificare il problema, i valori restituiti da u saranno interi.

1. Passo in input una rete (G,u,s,t) con $u : E(G) \to \mathbb{Z}_+$;
2. Pongo $f(e) := 0$ per ogni $e \in E(G)$;
3. Trovo un cammino *f-aumentante* P. Se non esiste vado al punto 5;
4. Calcolo $\gamma := \min_{e \in E(P)} u_e$. Aumento f lungo P di γ e torno al punto 3;
5. Restituisco il flusso s-t f, che sarà di valore massimo.

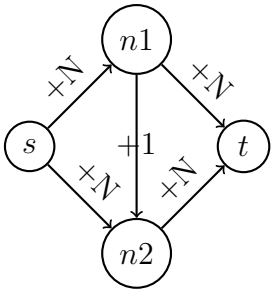

Figura 1:

Gli archi che danno il valore di γ al passo 4 sono spesso chiamati archi collo di bottiglia, o bottleneck.

La scelta di γ garantisce che f continui a essere un flusso. Dato che P è un cammino s-t, la legge della conservazione di flusso è verificata per tutti gli archi eccezion fatta per s e t.

È importante prestare attenzione a come si trova il cammino aumentante, in generale il procedimento è facile, ma in presenza di capacità irrazionali un'errata scelta degli archi potrebbe rendere il problema non risolvibile dato che l'algoritmo non termina o termina con risultato errato.

Inoltre, anche con valori interi, potrebbero volerci un numero esponenziale di aumenti, ad esempio considerando $N \in \mathbb{N}$ e il grafo rappresentato in figura 1, scegliendo sempre il cammino aumentante di lunghezza 3 termina in pari a $2N$.

Possiamo, grazie all'algoritmo di Ford-Fulkerson, dimostrare che un flusso s-t f è massimo se e solo se non esiste alcun cammino f-aumentante, e che quel valore f è uguale alla capacità minima di un taglio s-t. Inoltre, si può facilmente dimostrare il teorema del Flusso Intero, che esprime il semplice fatto che se le capacità della rete sono intereallora esiste un flusso massimo intero.

1.3 Algoritmo di Edmonds-Karp

Dato che l'algoritmo di Ford-Fulkerson potrebbe non terminare correttamente o terminare in tempo esponenziale, nel 1972 Jack Edmonds e Richard Karp hanno ottenuto il primo algoritmo tempo-polinomiale per il problema del flusso massimo, cambiando il passo 3. dell'algoritmo di Ford-Fulkerson.

1. Passo in input una rete (G, u, s, t)
2. Pongo $f(e) := 0$ per ogni $e \in E(G)$

3. Trovo un cammino f aumentante minimo P. Se non ne esiste nessuno vado al punto 5.

4. Calcolo $\gamma := \min_{e \in E(P)} u_e$. Aumento f lungo P di γ e torno al punto 3.

5. Restituisco il flusso s-t f, che sarà di valore massimo.

Si fa notare che il punto 3 dell'algoritmo si può implementare con una BFS. L'algoritmo di Edmonds-Karp termina, al massimo, dopo $\frac{mn}{2}$ aumenti, quindi, grazie a questo algoritmo, si può risolvere il problema del flusso massimo in tempo $O(m^2n)$, dove m è il numero di archi e n il numero di nodi.

1.4 Shortest Augmenting Path

Come suggerito da Orlin in [2], il problema potrebbe essere affrontato in maniera diversa, sempre seguendo l'algoritmo di Ford-Fulkerson, da questa premessa, nasce l'algoritmo Shortest Augmenting Path.

L'algoritmo di Shortest Augmenting Path aumenta sempre il valore del flusso attraverso il percorso più breve che collega il nodo sorgente s al nodo destinazione t del grafo dei residui. Un approccio naturale sarebbe l'esplorazione del grafo dei residui tramite BFS, ma ogni iterazione richiederebbe,nel caso peggiore, $O(m)$ passi , quindi con tempo finale di esecuzione $O(nm^2)$.

Questo tempo computazionale è eccessivo, ma possiamo migliorarlo: possiamo sfruttare il fatto che la distanza minima tra ogni nodo e il nodo destinazione t è monotonicamente non decrescente per tutte le iterazioni: sfruttando questa proprietà possiamo ridurre il tempo medio per ogni iterazione a $O(n)$, portando ad avere un tempo finale di risoluzioni $O(n^2m)$.

L'algoritmo Shortest Augmenting Path procede attraverso flussi aumentanti attraverso gli archi ammissibili, costruendo un cammino aumentate ammissibile aggiungendo un arco alla volta.

L'algoritmo mantiene un parziale cammino ammissibile, cioè un percorso da s a un generico nodo i fatto solamente di archi ammissibili, e iterativamente procedo con attraverso le istruzioni di *Advance* o di *Retreat* dal nodo corrente, quindi l'ultimo nodo del parziale cammino ammissibile. Se il nodo corrente i è incidente con un arco ammissibile (i, j), procediamo un'operazione di Advance e aggiungiamo l'arco (i, j) al parziale cammino ammissibile, altrimenti procediamo a con un'operazione di Retreat, e torniamo indietro di un arco. Ripetiamo questo procedimento finché il flusso non è massimo. Vediamo nel dettaglio il funzionamento dell'algoritmo:

Algoritmo *Shortest Augmenting Path*:

$x \leftarrow 0$

ottengo la distanza esatta $d(i)$, questa si può ottenere attraverso una BFS all'indietro a partire da t.
$i \leftarrow s$
while $d(s) < n$ **do**
  if i ha un arco ammissibile **then**
    $advance(i)$
    if $i = t$ **then**
      $augment$
      $i \leftarrow s$
    end if
  else
    $retreat(i)$
  end if
end while

Vediamo nello specifico le operazioni di advance, retreat e augment.

Procedura $advance(i)$
rendo (i, j) un arco ammissibile in $A(i)$
$\text{pred}(j) \leftarrow i$
$i \leftarrow j$

Procedura $retreat(i)$
$d(i) \leftarrow \min\{d(j) + 1 : (i, j) \in A(i) \wedge r_{ij} > 0\}$
if $i \neq s$ **then**
  $i \leftarrow pred(i)$
end if

Procedura augment
usando i predecessori identifico un cammino aumentante P dal nodo sorgente s al nodo destinazione t
$\delta \leftarrow \min\{r_{ij} : (i, j) \in P\}$
aumento di δ unità il flusso passanta attravverso P.

1.5 Possibili applicazioni

Il problema del flusso massimo e il problema del taglio minimo si possono presentare in un'ampia varietà di situazioni e in diverse forme, alcune volte si potrebbe presentare come un sotto-problema di un problema su rete più complesso, ad esempio il problema del problema di flusso a costo minimo o un problema di flusso generalizzato. Il problema si può presentare anche direttamente come un problema di machine scheduling, assegnamento di moduli informatici ai processi del computer, arrotondamento dei dati del censimento per mantenere la riservatezza del singolo nucleo familiare,

programmazione delle navi cisterna oppure un problema di flusso dinamico massimo, dove, oltre alla capacità di un arco, bisogna la variabile aggiungere il tempo, che indica il tempo necessario per passare quell'arco.

Capitolo 2

Ottimizzazioni usate con implementazione monodirezionale

2.1 Nessuna ottimizzazione

L'algoritmo usato è semplicemente quello di Edmonds-Karp, senza alcuna ottimizzazione. Vediamo come sono state costruite le strutture dati necessarie:

2.1.1 Strutture dati

MonoEdge

Rappresenta l'arco, caratterizzato dalle seguenti proprietà:

- **Next**, rappresenta il nodo dove entra l'arco;
- **Capacity**, rappresenta la capacità residua dell'arco;
- **Flow**, rappresetna il flusso inviato dell'arco.

Per ogni arco MonoEdge si va a creare l'arco inverso, ossia **ReversedMonoEdge**, che ha le stessa caratteridtiche di MonoEdge,ma Next è il nodo da cui l'arco esce.

Node

Rappresenta il nodo, caratterizzato dalle seguenti proprietà:

- **Name** rappresenta il nome dato al nodo.
- **Next**, è la lista contenente gli archi MonoEdge e ReversedMonoEdge che partono dal nodo;

- **Label**, rappresenta la distanza, cioè il numero minimo di nodi da attraversare, tra il nodo e il nodo sorgente s;
- **PreviousNode**, mi serve per indicare il percorso da fare, indicandomi il nodo precedente da utilizzare per poter raggiungere il nodo sorgente s;
- **FlussoPassante**, indica la quantità di flusso che, con le informazioni ricevute fino al dato nodo, è possibile inviare attraverso il percorso descritto attraverso i PreviousNode.

Tramite il proprio metodo **SendFlow**,dati in input la quantità di flusso da inviare e un nodo collegato al nodo chiamante, può inviare il dato flusso negli archi MonoEdge e ReversedMonoEdge che collegano i due nodi.

Graph

Rappresenta il grafo, è un insieme contenente i nodi.

2.1.2 Descrizione

Data una rete (G, u, s, t), cerco un cammino aumentante:

Per ogni nodo del grafo, tranne il nodo sorgente s, cancello tutte le informazioni di indirizzamento, quindi PreviousNode, FlussoPassante e Label. Partendo dal nodo sorgente s, cerco un cammino aumentante utilizzando una BFS, quindi utilizzo una coda inizializzata inserendovi il nodo s al suo interno. Per ogni nodo estratto dalla coda utilizzata per la BFS, esploro gli archi contenuti in **Next**, l'arco deve essere o MonoEdge con capacità positiva o ReversedMonoEdge con flusso inviato positivo, inoltre il nodo indicato dall'arco non deve essere stato esplorato precedentemente.

Se il nodo è esplorabile, aggiorno i suoi dati di indirizzamento, quindi PreviousNode, Label e FlussoPassante, con le informazioni date dal nodo esplorante e dall'arco che li collega.

Una volta trovato il percorso aumentante, invio il flusso indicatomi dal flusso passante del nodo destinazione t nel percorso descritto dai vari PreviousNode. Ripeto finché riesco a trovare dei cammini aumentanti.

Per ulteriori dettagli si invita alla visione degli pseudo-codici 1 e 2 presenti in appendice A.1

2.2 Ottimizzazione sugli ultimi livelli

Durante l'invio del flusso, ho un almeno un arco che diventa saturo, cioè che ha capacità residua pari a 0.

Posso non cancellare le informazioni dei nodi con label minore rispetto al nodo successore dell'arco saturo.

Inoltre, possiamo provare a riparare il nodo non più raggiungibile dato per la saturazione dell'arco, quindi cercare un nodo precedessore adeguato, nel caso riesca a riparare tutti i nodi che sono diventati irraggiungibili nell'ultima iterazione, posso confermare di aver trovato un nuovo cammino.

2.2.1 Strutture dati

BiEdge

Rappresenta l'arco, è caratterizzato dalle seguenti proprietà:

- **NextNode**, rappresenta il nodo dove l'arco entra;
- **PreviousNode**, rappresenta il nodo dove l'arco esce;
- **Flow**, rappresenta la quantità di flusso che passa attraverso quell'arco;
- **Capacity**, rappresenta la capacità residua dell'arco;
- **Reversed**, indica come deve essere letto l'arco, quindi se deve considerato come l'arco inverso, e quindi, se durante l'invio del flusso devo inviarlo o ritirarlo.

Node

Rappresenta il nodo, è caratterizzato dalle seguenti proprietà:

- **Name**, rappresenta il nome dato al nodo;
- **Edges**, è la lista degli archi incidente al nodo;
- **Label**, rappresenta la distanza dal nodo sorgente s;
- **PreviousNode**, rappresenta il nodo predecessore, cioè il nodo che è l'ha esplorarlo;
- **PreviousEdge**, rappresenta l'arco che collega il nodo a PreviousNode;
- **Visited**, mi indica se il nodo è stato visitato, quindi se da quel nodo è possibile arrivare a s;

- **Valid**, mi indica se PreviousEdge è un arco saturo, e quindi, se è ancora possibile raggiungere PreviousNode.

Tramite il proprio metodo **SendFlow**, dato in input la quantità di flusso da inviare, invio, o ritiro nel caso $Reversed = true$, quella certa quantità di flusso dall'arco PreviousEdge, notificando se l'arco si è saturato o meno.

Graph

Rappresenta il grafo, i suoi nodi sono contenuti come segue:

- **LabeledNode**, implementata tramite una lista di insieme, raccoglie in ogni insieme i nodi con il valore specifica Label, corrispondente alla posizione nella lista.
- **InvalidNode**, insieme di nodi contenente i nodi che non è stato possibile riparare.

I metodi di questa classe riguardano solo lo spostamento di nodi tra gli insiemi, quindi, o tra InvalidNode ed un insieme in LabeledNode; oppure all'interno di LabeledNode, quando un nodo, in seguito a una nuova esplorazione, deve cambiare Label.

2.2.2 Descrizione

Data una rete (G, u, s, t), cerco un cammino aumentante.

Una volta trovato il cammino aumentante, ricavo il flusso inviabile e, partendo dal nodo destinazione t, retrocedo attraverso PreviousNode verso il nodo sorgente s, inviando (o ritirando, nel caso $Reversed = true$) il flusso dell'arco PreviousEdge.

Nel caso in cui, inviando il flusso, un arco si saturi, cioè la sua capacità residua si azzeri, inserisco il nodo che lo ha come PreviousNode in una pila di nodi.

Proseguo finché non posso trovare più altri cammini aumentanti.

Ricerca del cammino aumentante

Ricevo in input la rete (G, u, s, t), e la pila $malati$, contenente i nodi non più raggiungibili dopo l'ultima iterazione.

Nel caso la pila sia vuota, cancello da ogni nodo del grafo, tranne il nodo sorgente s, le informazioni di indirizzamento, quindi Visited, PreviousNode e PreviousEdge; inoltre inserisco s nella coda utilizzata per esplorare il grafo.

Nel caso la pila non sia vuota, provo a riparare ogni nodo contenuto nella pila: nel caso trovi un nodo non riparabile, lo salvo in una variabile e non procedo a riparare gli altri, se, invece, riesco a riparare tutti i nodi della pila, indico che ho trovato un nuovo cammino aumentante.

Nel caso non sia riuscito a riparare un nodo, procedo a inizializzare la coda inserendovi tutti i nodi che hanno label di uno precedente rispetto alla label del nodo non riparabile, e cancello le informazioni di indirizzamento dei nodi che hanno label pari o superiore a quella del nodo non riparabile.

Successivamente, procedo a esplorare il grafo :

1. dalla coda estraggo un nodo *element*;
2. per ogni arco uscente da *element*, se il nodo dove l'arco entra non è già stato esplorato e la capacità residua dell'arco è positiva, aggiorno le informazioni di indirizzamento e la label e, se non è t, lo aggiungo alla coda;
3. per ogni arco entrante in *element*, se il nodo dove l'arco esce non è già stato esplorato e il flusso dell'arco è positivo, lo aggiorno e, se non è t, lo aggiorno alla coda.

Procedo a esplorare i nodi del grafo finché non trovo t, questo mi indica che ho trovato un cammino aumentante.

Riparare un nodo

Quando, nella descrizione precedente, si diceva di poter riparare un nodo n, si indicava cercare un nodo con le seguenti caratteristiche:

- il nodo deve essere adiacente a n;
- il nodo deve risultare valido e visitato, quindi da quel nodo deve essere possibile raggiungere s;
- la label del nodo deve essere di uno inferiore alla label di n
- nel caso l'arco che collega i due nodi esca da n, il flusso dell'arco deve essere positivo, altrimenti, se l'arco entra in n, la capacità residua deve essere positiva;

Se il nodo rispetta questi requisiti, posso aggiornare le informazioni di indirizzamento di n, altrimenti lo indico come non valido, inserendolo in InvalidNode.

Per ulteriori dettagli si invita alla visione degli pseudo-codici 3,4 e 5 presenti nell'appendice A.2.

2.3 Propagazione della malattia

Data l'idea dell'ottimizzazione sugli ultimi livelli, è possibile svilupparla e, invece di esplorare tutti i nodi a partire dall'insieme contenente il primo nodo non riparabile, posso usare quel nodo per espandere la malattia, quindi indicare i nodi più raggiungibili, dato che erano stati esplorati da un nodo ora malato.

Per ogni nodo malato, esploro i nodi da lui scoperti , provo a ripararli, nel caso uno di questi nodi non sia riparabile, significa che non è più valido, e quindi che, con lui, devo proseguire la propagazione della malattia.

Una volta conclusa la propagazione della malattia per tutti i nodi ho due possibili scenari: la propagazione della malattia ha raggiunto t, e non sono riuscito a ripararlo, in tal caso dovrò esplorare il grafo; oppure la propagazione della malattia non ha raggiunto t o, se lo ha raggiunto, sono riuscito a ripararlo, quindi ho già ottenuto un cammino aumentante.

2.3.1 Strutture dati

Le strutture dati sono le stesse usate in Ottimizzazione negli ultimi livelli (vedesi 2.2.1).

2.3.2 Descrizione

Data una rete (G, u, s, t), cerco un cammino aumentate.

Se lo trovo, ricavo il valore del flusso inviabile, partendo da t e retrocedendo verso s, valutando il massimo flusso inviabile, quindi cercando il valore di capacità minimo (dove l'arco ha $Reversed = true$, valuto il flusso).

Trovato il flusso inviabile, procedo a inviarlo nel percorso descritto dai PreviousNode di ogni nodo, inviando (o richiedendo, nel caso $Reversed = true$) il flusso indicato, se durante l'invio del flusso un arco diventa saturo, vado a inserire in una pila il nodo che lo ha come PreviousEdge.

Ripeto finché non riesco più a trovare un cammino aumentante.

Ricerca del cammino aumentante

Ricevo in input la rete (G, u, s, t) e la pila di nodi con PreviousEdge saturo.

Se la pila è vuota, significa che è la prima iterazione, quindi procedo a inserire nella coda che utilizzerò per esplorare il grafo, qui chiamata *coda*, il nodo sorgente s, e salvo il fatto è la prima iterazione.

Se la pila non è vuota, cerco di riparare tutti i nodi al suo interno: se riesco a ripararli tutti, significa che ho un nuovo cammino aumentante, altrimenti, per ogni nodo che non sono riuscito a riparare, lo inserisco nella coda *malati*.

Salvo il primo nodo di *malati*, qui chiamato $firstNoCap$ e proseguo con la propagazione della malattia : per ogni nodo contenuto nella coda *malati* proseguo come segue:

1. inserisco il nodo in una coda *propagazione*
2. ripeto i seguenti passaggi finché *propagazione* non è vuota
3. estraggo un nodo da *propagazione* e cerco di ripararlo
4. nel caso non riesca a ripararlo, inserisco i nodi adiacenti che lo hanno come PreviousNode in *propagazione*
5. nel caso sia riuscito a ripararlo, nel caso sia il nodo t, lo restituisco, altrimenti lo inserisco in *coda*

Se, dopo che *malati* si è svuotata, il nodo destinazione t è valido, indico che ho trovato un cammino aumentante, e lo restituisco, altrimenti cancello le informazioni dei nodi con label maggiore o pari a quella di $firstNoCap$, e, nel caso la coda sia vuota, vi inserisco tutti i nodi con label di uno precedente a quella di $firstNoCap$.

Proseguo con l'esplorazione dei nodi : finché *coda* non è vuota, estraggo un nodo *element*, e analizzo i suoi archi:

Per ogni arco entrante in *element*, controllo che il flusso sia positivo e che, o il nodo da cui l'arco esce non sia valido o che abbia label maggiore rispetto a quella di *element* o che sia la prima iterazione;

Per ogni arco uscente da *element* controllo che la capacità residua sia positiva e che o il nodo in cui l'arco entra non sia valido o che abbia label maggiore rispetto a quella di *element* o che sia la prima iterazione;

Se l'arco soddisfa queste richieste aggiorno le sue informazioni e quelle del nodo a lui collegato (che non è *element*), oltre a inserirlo in *coda*.

Riparare un nodo

Riparare un nodo significa cercare tra i suoi nodi adiacenti un nodo che soddisdi i seguenti requisiti:

- abbia label di uno inferiore alla sua
- sia valido
- sia visitato
- se l'arco entra nel nodo da riparare, la capacità residua di quell'arco deve essere positiva, se l'arco esce dal nodo da riparare, il flusso di quell'arco deve essere positivo.

Nel caso trovi un nodo con questi requisiti, aggiorno le informazioni del nodo da riparare, altrimenti lo indico come non valido, non esplorato e lo inserisco nell'insieme InvalidNode.

Per maggiore dettagli, si inviata alla visione dello pseudo-codice 3, 6 e 7 in appendice A.2.

2.4 Shortest Augmentng Path

2.4.1 Strutture dati

BiEdge

Rappresenta l'arco, è costruito come quello presentato in Ottimizzazione sugli ultimi livelli (vedesi capitolo 2.2.1).

Node

Rappresenta il nodo è rappresentato dalle seguenti proprietà:

- **Name**, rappresenta il nome dato al nodo;
- **Edges**, è la lista di archi incidenti al nodo;
- **Distance**, rappresenta la distanza tra quel nodo e t;
- **PreviousNode**, rappresenta il nodo predecessore, cioè il nodo che l'ha esplorato, indicandomi il percorso per raggiungere s;
- **PreviousEdge**, rappresenta l'arco che collega il nodo a PreviousNode.

Graph

Rappresenta il grafo, è un insieme di nodi.

2.4.2 Descrizione

L'algoritmo si divide in tre pari:

- **Augment**, dopo aver trovato un cammino aumentante, invio il flusso;
- **Advance**, ho trovato un nuovo nodo e avanzo usando quel nodo per l'iterazione successiva;
- **Retreat**, se non ho trovato un nodo che mi permette di eseguire un Advance, ricalcolo la distanza del nodo esplorato e, se possibile, ritorno al nodo precedente.

Dato la sua grandezza, con una ricorsione si sarebbe andati incontro a uno stack overflow, quindi ho preferito renderla iterativa.

Dato una rete (G, u, s, t), eseguo una bfs da t verso s per calcolare la distanza esatta di tutti i nodi e invio il flusso trovato, salvandomi la quantità di flusso inviato in una

variabile $fMax$. Finché la distanza del nodo sorgente s è minore rispetto al numero di nodi del grafo, procedo come segue: cerco un cammino aumentante e,con esso, la quantità di flusso inviabile f; se f è pari a 0, interrompo l'esecuzione e restituisco il valore di flusso inviato $fMax$, altrimenti aumento $fMax$ di f e procedo con l'azione di Augment, e cancello le informazioni di indirizzamento, quindi PreviousNode e PreviousEdge, per ogni nodo esplorato.

Al termine, restituisco la quantità di flusso inviata $fMax$.

Augment

Dato la quantità di flusso f da inviare e la rete (G, u, s, t), dal nodo destinazione t, invio f nell'arco indicato in PreviousEdge, e procedo nel nodo indicato in PreviousNode.

Ripeto queste azioni finché il nodo esplorato è s.

Esplorazione del grafo

Al contrario dei precedenti algoritmi, qui procedo tramite Dfs: Inizializzo una pila per l'esplorazione e vi inserisco la coppia $(s, +\infty)$. Finché la pila non è vuota procedo come segue:

Estraggo la coppia nodo valore, e finché il nodo estratto ha una distanza minore rispetto a $\#V(G)$, esploro i nodi a lui adiacenti, se trovo un nodo che ha distanza di uno inferiore alla sua, con l'arco che li collega con capacità positiva, procedo con l'operazione di Advance:

Aggiorno i dati del nodo scoperto, aggiorno la quantità di flusso inviabile, ottenendo la minima tra la capacità dell'arco che collega i due nodi e quella data dalla pila; se il nodo esplorato è il nodo destinazione t, restituisco il valore del flusso trovato, al contrario inserisco nella pila il nodo trovato e la quantità di flusso inviabile fino a quel momento.

Se non ho trovato alcun nodo con le precedenti caratteristiche, procedo con l'operazione di Retreat: inizializzo una variabile min a $+\infty$, e analizzo gli archi uscenti dal nodo che sto esplorando: se trovo un arco che ha capacità positiva, salvo in min il valore minimo tra la distanza del nodo dove entra l'arco e min stessa. Aggiorno la distanza del nodo esplorato con $min + 1$ e inserisco nella pila il valore di flusso prima dato dalla pila e, se il nodo che sto esplorando non è s, il nodo predecessore, altrimenti s stesso.

Se non ho più nodi nella pila o se la distanza del nodo esplorato è maggiore rispetto a $\#V(G)$, restituisco il valore 0.

Per maggiore dettagli, si inviata alla visione degli pseudo-codici 8 e 9 presentati in appendice A.3

Capitolo 3

Ottimizzazioni usate con implementazione bidirezionale

Gli algoritmi e le ottimizzazioni utilizzate per la risoluzione problema del flusso massimo sono gli stessi presentati nel capitolo precedente, inoltre, dato la bidirezionalità, si sono scelte alcuni metodi per poter esplorare il grafo, qui di seguito presentati:

3.1 Presentazione delle strategie usate per l'esplorazione bidirezionale

Sono state applicate tre diverse strategie per l'esplorazione dei nodi data la bidirezionalità :

- **Label**: per ogni parte esploro i nodi procedendo con pari label
- **NodeCount**: esploro un nodo alla volta da una parte, per procere con un nuovo nodo dall'altra parte
- **NodePropagation**: esploro tutti i nodi adiacenti al nodo esaminato di una certa parte, per poi procedere dall'altra parte

Andiamo a vedere un po' più nello specifico come avviene l'esplorazione attraverso queste strategie.

Con "aggiorno le sue informazioni", si intende aggiornare le seguenti proprietà :

- validità
- se è stato scoperto partendo dal nodo sorgente s o dal nodo destinazione t
- la sua Label, quindi la distanza da s o da t, a seconda di chi l'ha scoperto

- informazioni per l'indirizzamento, quindi previousNode e previousEdge per i nodi esplorati da s, nextNode e nextEdge per i nodi esplorati da t, entrambi per i nodi di confine
- se l'arco deve considerarsi Reversed, quindi se durante l'invio del flusso deve inviare o richiedere flusso

3.1.1 Esplorazione per Label

Date le due code *codaSource*, contenente i nodi da cui far partire l'esplorazione dalla parte di s, e *codaSink*, contenente i nodi da cui far partire l'esplorazione dalla parte di t, ne creo una terza, $codaBuffer$, che verrà usata come buffer.
Finché entrambe le code sono vuote, procedo come segue:

1. finché *codaSource* non è vuota procedo come segue
   (a) estraggo un nodo *element* da *codaSource*
   (b) se è stato scoperto da t, se non è stato esplorato o se non è valido, lo salto e procedo con un nuovo nodo
   (c) esploro gli archi che escono da *element* con capacità positiva
   (d) se il nodo dove l'arco entra risulta visitato ed è stato scoperto precedentemente da t, questo mi indica che ho trovato un cammino aumentante, aggiorno i valori del nodo trovato, lo inserisco in un insieme di nodi di confine e lo restituisco
   (e) se il nodo dove l'arco entra risulta non visitato ed è non è stato precedentemente scoperto da t, aggiorno le sue informazioni e lo inserisco in $codaBuffer$
   (f) esploro gli archi che entrano in *element* con flusso positivo
   (g) se il nodo dove esce l'arco risulta visitato ed è stato scoperto da t, questo mi indica che ho trovato un nuovo cammino aumentante, aggiorno le informazioni del nodo dove esce l'arco, lo inserisco in un insieme di nodi di confine e lo restituisco
   (h) se il nodo dove esce l'arco risulta non visitato e non è stato precedentemente scoperto da t, aggiorno le sue informazioni e lo inserisco in $codaBuffer$
2. procedo a scambiare i puntatori di *codaSource* e di $codaBuffer$
3. finché *codaSink* non è vuota procedo come segue
   (a) estraggo un nodo *element* da *codaSink*

(b) se risulta scoperto da s, se risulta non visitato o se risulta non valido, lo salto e procedo con un nuovo nodo

(c) esploro gli archi entranti in *element* con capacità positiva

(d) se il nodo dove l'arco esce è stato visitato, controllo se è stato scoperto da t, in tal caso procedo all'arco successivo

(e) se il nodo dove l'arco esce è stato visitato da s, indico che ho trovato un nuovo cammino aumentante, aggiorno i dati di *element*, lo inserisco in un insieme di nodi di confine e lo restituisco

(f) se il nodo dove l'arco esce non è stato visitato precedentemente, aggiorno le sue informazioni e lo inserisco in $codaBuffer$

(g) esploro gli archi uscenti da *element* con flusso positivo

(h) se il nodo dove l'arco entra è stato visitato, controllo se è stato scoperto da t, in tal caso procedo all'arco successivo

(i) se il nodo dove l'arco entra è stato visitato da s, indico che ho trovato un nuovo cammino aumentante, aggiorno i dati di *element*, lo inserisco in un insieme di nodi di confine e lo restituisco

(j) se il nodo dove l'arco entra non è stato visitato precedentemente, aggiorno le sue informazioni e lo inserisco in $codaBuffer$

4. procedo a scambiare i puntatori di $codaSink$ e $codaBuffer$

Per maggiore dettagli si invita alla visione dello pseudo-codice 10 presente nell'appendice A.4

3.1.2 Esplorazione esplorando un nodo alla volta

Date le informazioni sulle parti del grafo dove dovrò lavorare e le due code *codaSource*, contenente i nodi da cui far partire l'esplorazione dalla parte di s, e *codaSink*, contenente i nodi da cui far partire l'esplorazione dalla parte di t, ne creo altre due: *codaEdgeSource*, che conterrà gli archi da esplorare dell'elemento estratto da *codaSource*, e *codaEdgeSink*, che conterrà gli archi da esplorare dell'elemento estratto da *codaSink*.

Finché sia *codaSource* sia *codaSink* non sono entrambe vuote, proseguo come segue

1. controllo che *codaSource* non sia vuota e o la coda *codaEdgeSource* sia vuota, o che non sia possibile inserire elementi in *codaEdgeSink*,nel caso si rispetto queste richieste proseguo come segue, altrimenti vado alla riga 4;

2. estraggo un elemento *elementSource* da *codaSource*, tale elemento deve risultare esplorato a partire da s,visitato e valido, altrimenti procedo con l'estrazione;

3. inserisco in *codaEdgeSource* tutti gli archi che hanno come un estremo *elementSource* e l'altro estremo un nodo esplorabile ;

4. controllo che che *codaSink* non sia vuota e o la coda *codaEdgeSink* sia vuota, o che non sia possibile inserire elementi in *codaEdgeSource*, nel caso si rispetto queste richieste proseguo come segue, altrimenti vado alla riga 7;

5. estraggo un elemento *elementSink* da *codaSink*, tale elemento deve risultare esplorato a partire da t,visitato e valido, altrimenti procedo con l'estrazione;

6. inserisco in *codaEdgeSink* tutti gli archi che hanno come un estremo *elementSink* e l'altro estremo un nodo esplorabile;

7. se, compatibilmente con la possibilità delle loro parti di essere esplorate, *codaEdgeSource* e *codaEdgeSink* non sono vuote, procedo come segue

   (a) se *codaEdgeSource* non è vuota, estraggo un arco *edgeSource*, altrimenti vado al punto 7h

   (b) se il nodo dove esce *edgeSource* è *elementSource* e la capacità di *edgeSource* è positiva,procedo come segue, altrimenti vado al punto 7e

   (c) se il nodo dove *edgeSource* entra risulta visitato, ed è stato scoperto da t, indico che ho trovato un cammino, aggiorno le informazioni nel nodo, lo aggiungo ai nodi di confine e lo restituisco

(d) se il nodo dove *edgeSource* entra risulta come non visitato ed è stato scoperto da s procedo ad aggiornare le sue informazioni e lo inserisco in *codaSource* e procedo dal punto 7h

(e) se il nodo dove entra *edgeSource* è *elementSource* e il flusso di *edgeSource* è positivo, allora precedo come segue, altrimenti vado al punto 7h

(f) se il nodo dove esce *edgeSource* risulta visitato, ed è stato scoperto da t, indico che ho trovato un cammino, aggiorno le informazioni nel nodo, lo aggiungo ai nodi di confine e lo restituisco

(g) se il nodo dove esce *edgeSource* risulta come non visitato ed è stato scoperto da s procedo ad aggiornare le sue informazioni e lo inserisco in *codaSource*

(h) se *codaEdgeSink* non è vuota, estraggo un arco *edgeSink* vado al punto 7

(i) se il nodo dove entra *edgeSink* è *elementSink* e la capacità di *edgeSink* è positiva, procedo, altrimenti vado al punto 7l

(j) se il nodo dove esce *edgeSink* risulta visitato, controllo da che parte è stato scoperto, nel caso sia stato scoperto da t, torno a 7, altrimenti ho trovato un cammino, aggiorno i valori di *elementSink*, lo aggiungo ai nodi di confine e lo restituisco

(k) se il nodo dove esce *edgeSink* non risulta visitato, aggiorno le sue informazioni, lo aggiungo alla coda *codaSink* e torno a 7

(l) se il nodo dove esce *edgeSink* è *elementSink* e il flusso di *edgeSink* è positivo, procedo, altrimenti vado a 7

(m) se il nodo dove entra *edgeSink* risulta visitato, controllo da che parte è stato scoperto, nel caso sia stato scoperto da t, torno a 7, altrimenti ho trovato un cammino, aggiorno i valori di *elementSink*, lo aggiungo ai nodi di confine e lo restituisco

(n) se il nodo dove entra *edgeSink*n risulta visitato, aggiorno le sue informazioni e lo aggiungo alla coda *codaSink*

Per maggiori dettagli si invita alla visione dello pseudo-codice 11, presente nell'appendice A.4

3.1.3 Esplorazione propagando i nodi

Date le informazioni sulle parti del grafo dove dovrò lavorare e le due code *codaSource*, contenente i nodi da cui far partire l'esplorazione dalla parte di s, e *codaSink*, contenente i nodi da cui far partire l'esplorazione dalla parte di t, procedo come segue per esplorare il grafo:

1. Finché entrambe le code non sono vuote, procedo come segue:
   (a) se *codaSource* non è vuota, procedo, altrimenti vado al punto 1i
   (b) estraggo un nodo *elementSource* da *codaSource*, tale nodo deve essere stato scoperto da s, deve essere valido e deve risultare visitato
   (c) esploro gli archi uscenti da *elementSource* con capacità positiva
   (d) se il nodo da in cui entra l'arco è stato scoperto da source, non è stato esplorato o risulta essere non valido, aggiorno i suoi dati e lo accodo a *codaSource*
   (e) se il nodo in cui entra l'arco è stato visitato, è stato scoperto da t ed è valido, ho scoperto un nuovo cammino aumentante, aggiorno i dati del nodo, lo aggiungo ai nodi di confine e lo restituisco
   (f) esploro gli archi entranti in *elementSource* con flusso positivo
   (g) se il nodo da in cui esce l'arco è stato scoperto da s, non risulta esplorato o risulta essere non valido, aggiorno i suoi dati e lo accodo a *codaSource*
   (h) se il nodo da cui esce l'arco risulta visitato, è stato scoperto da t ed è valido, ho scoperto un nuovo cammino aumentante, aggiorno i dati del nodo, lo aggiungo ai nodi di confine e lo restituisco
   (i) se *codaSink* non è vuota,procedo, torno al punto 1
   (j) estraggo un nodo *elementSink* da *codaSink*, tale nodo deve essere stato scoperto da t, deve essere valido e deve essere risultare visitato
   (k) esploro gli archi entranti in *elementSink* con capacità positiva
   (l) se il nodo in cui l'arco esce è stato scoperto da t, non è risulta esplorato o risulta essere non valido, aggiorno i suoi dati e lo accodo a *codaSink*
   (m) se il nodo da cui esce l'arco risulta visitato, è stato scoperto da s ed è valido, ho scoperto un nuovo cammino aumentante, aggiorno i dati di *elementSink*, lo aggiungo ai nodi di confine e lo restituisco
   (n) esploro gli archi uscente da *elementSink* con flusso positivo
   (o) se il nodo in cui l'arco entra è stato scoperto da t, non risulta esplorato o o risulta essere non valido, aggiorno i suoi dati e lo accodo a *codaSink*

(p) se il nodo da cui entra l'arco risulta visitato, è stato scoperto da s ed è valido, ho scoperto un nuovo cammino aumentante, aggiorno i dati di $elementSink$, lo aggiungo ai nodi di confine e lo restituisco

Per maggiori dettagli si invita alla visione dello pseudo-codice 12, presente nell'appendice A.4

3.2 Nessuna ottimizzazione

3.2.1 Strutture dati utilizzate

BiEdge

Rappresenta l'arco, è rappresentato dalle seguenti proprietà :

- **NextNode**, rappresenta il nodo dove entra l'arco;
- **PreviousNode**, rappresenta il nodo dove esce l'arco;
- **Capacity**, rappresenta la capacità residua dell'arco, nello pseudo-codice è rappresentata anche come funzione u_f;
- **Flow**, rappresenta il flusso inviato dell'arco, nello pseudo-codice è rappresentata anche come funzione f;
- **Reversed**, rappresenta come va letto l'arco, deve essere letto come (PreviousNode, NextNode) nel caso sia false, nel caso sia true va letto come l'arco inverso (NextNode,PreviousNode).

Tramite il proprio metodo AddFlow, riesco ad aggiornare i dati, inviando (o richiedendo, nel caso Reversed sia true) il flusso richiesto nell'arco chiamato.

Node

Rappresenta il nodo, è rappresentato dalle seguenti proprietà

- **Name** rappresenta il nome dato al nodo.
- **Visited**, rappresenta se il nodo risulta visitato, e quindi se è possibile raggiungere s o t attraverso quel nodo;
- **SourceSide**, rappresenta da chi è stato scoperto, true se è stato scoperto da s, false altrimenti;
- **Label**, rappresenta la distanza tra il nodo e s
- **Edges**, è la lista di archi a lui incidenti, nello pseudo-codice si indica con le funzioni δ, δ^+ per gli archi uscenti e δ^-, per gli archi entranti;
- **PreviousNode**, rappresenta il nodo da cui è stato esplorato, e quindi il nodo che posso usare per tornare al nodo sorgente s, si fa notare che è usato solo da nodi di confine e da nodi con SourceSide vero;

- **PreviousEdge**, rappresenta l'arco che collega il nodo al proprio PreviousNode;
- **NextNode**, rappresenta il nodo da cui è stato esplorato, e quindi il nodo che posso usare per tornare al nodo destinazione t, si fa notare che è usato solo con SourceSide falso;
- **NextEdge**, rappresenta l'arco che collega il nodo e NextNode.

I suoi metodi sono solo per inizializzazione e per impostare il valore delle sue proprietà.

Graph

Rappresenta il grafo, è rappresentato dai due insiemi :

- **SourceNodes**, insieme di nodi esplorati da Source
- **SinkNodes**, insieme di nodi esplorati da Sink

I suoi metodi riguardano i seguenti :

- inizializzazione, permenttendomi di aggiungere nodi agli insiemi;
- cambio di insieme di appartenza, permettendo di spostare un nodo dall'insieme SourceNodes all'insieme SinkNodes
- funzione reset, per ogni nodo di un dato insieme, li indico come non visitati, tranne per i nodi s e t

3.2.2 Descrizione

Data una rete (G, u, s, t), cerco un cammino aumentante, indicando quale parte del grafo devo esplorare (nella prima iterazione devo esplorare sia la parte di source sia quella di sink), se non lo trovo indico che non è possibile trovare nuovi cammini e termino l'esecuzione, ritornando il valore del flusso totale inviato.

Se trovo un cammino aumentante, calcolo il flusso inviabile attraverso il percorso descritto e invio il flusso trovato; quando incontro un arco che si satura durante l'invio del flusso, seleziono in quale parte del grafo si trova, così da esplorarla durante la nuova esecuzione.

Ricerca di un cammino aumentante

Ricevo in input la rete (G, u, s, t) e i due booleani *resetSource* e *resetSink* che mi indicano, rispettivamente, se devo esplorare la parte esplorata da s e/o se devo esplorare la parte esplorata da t.

Se *resetSource* è vero, indico per ogni nodo che è stato esplorato da s, tranne per s stesso, che non è stato visitato, e inserisco in *codaSource*, cioè la coda usata per l'esplorazione della parte Source, il nodo sorgente s. Se *resetSink* è vero, indico per ogni nodo che è stato esplorato da t, tranne per t stesso, che non è stato visitato, e inserisco in *codaSink*, cioè la coda usata per l'esplorazione della parte Sink, il nodo destinazione t. Successivamente procedo con l'esplorazione a seconda della strategia di esplorazione desiderata presentate nel capitolo 3.1.

Per ulteriori dettagli si invita alla visione degli pseudo-codici 13 e 14, presenti in appendice A.5

3.3 Ottimizzazione negli ultimi livelli

3.3.1 Strutture dati

BiEdge

Rappresenta l'arco, è come quello presentato nel capitolo 3.2.1

Node

Rappresenta il nodo, è rappresentato come quello presentato nel capitolo 3.2.1, ma con una proprietà dal funzionamento differente e una nuova proprietà:

- **Label**, rappresenta la distanza tra il nodo e, a seconda da chi l'ha scoperto, s o t;
- **Valid**, nuova proprietà, mi indica se l'arco che lo ha esplorato è saturo, quindi se è possibile raggiungere il nodo da cui è stato esplorato.

I suoi metodi sono solo per inizializzazione e per impostare il valore delle sue proprietà.

Graph

Rappresenta il grafo, è rappresentato dalle seguenti proprietà:

- **LabeledNodeSourceSide**, è una lista di insiemi, ogni insieme contiene i nodi, esplorati da s, con una certa label, il valore di quella label è uguale alla posizione che ha l'insieme nella lista;
- **LabeledNodeSinkSide**, è una lista di insiemi, ogni insieme contiene i nodi, esplorati da t, con una certa label, il valore di quella label è uguale alla posizione che ha l'insieme nella lista;
- **LastNodesSinkSide**, è un insieme di nodi, ci sono contenuti i nodi, definiti anche come nodi di confine, che, seppur esplorati in origine da t, sono stati esplorati anche da s, trovando così un cammino aumentante.
- **LastNodesSourceSide**, è un insieme di nodi, ci sono contenuti i nodi, esplorati da s, che sono adiacenti ai nodi di confine.

I metodi che riguardano Graph riguardano :

- inserimento di nodi in appositi insiemi, ad esempio **AddLast**, che va a inserire il nodo indicato in LastNodesSinkSide, e i suoi nodi adiacenti, esplorati da source, in LastNodesSourceSide.

- spostamento di da un insieme a un altro insieme, ad esempio **ChangeLabel**, che mi permette di spostare un nodo n, da un insieme dato dal suo SourceSide e dalla sua label, a un nuovo insieme, dettato dalle variabili date in input.

3.3.2 Descrizione

Data una rete (G, u, s, t), inizializzo a zero una variabile $fMax$ che mi terrà conto del valore del flusso inviato, e inizializzo le due pile *vuotiSource*, che conterrà i nodi non validi della parte di Source, e *vuotiSink*, che conterrà i nodi non validi della parte di Sink, inserendovi rispettivamente i nodi s e t.

Ciclicamente, cerco un cammino aumentante, se non lo trovo, restituisco il valore $fMax$ e termino l'esecuzione, altrimenti cerco il flusso inviabile attraverso quel cammino, svuoto le due pile *vuotiSource* e *vuotiSink*, poi procedo a inviare il flusso attraverso il cammino indicato, se un arco si satura, vado a inserire il nodo che lo ha come PreviousEdge o come NextEdge nell'apposita pila, aggiornando $fMax$ col valore del flusso inviabile trovato.

Ricerca del cammino aumentante

Ricevo in input la rete (G, u, s, t) e le due pile *vuotiSource*, contenente i nodi non validi di Source, e *vuotiSink*, contenente i nodi non validi di Sink. Creo le due code *codaSource* e *codaSink* per l'esplorazione dei nodi.

Se *vuotiSource* non è vuota, provo a riparare tutti i nodi ivi contenuti: se riesco a ripararli tutti e *vuotiSink* è vuota, cerco tra i nodi di confine un nodo visitato e valido che riesca a raggiungere s.

Altrimenti, al primo nodo non riparabile, mi fermo, salvo quel nodo, lo inserisco nuovamente in *vuotiSource*, riempio *codaSource* come segue:

- se il nodo è s, lo inserisco in *codaSource*
- se è un nodo di confine, inserisco i nodi contenuti in LastNodesSourceSide in *codaSource*
- altrimenti indico come non visitati i nodi con label pari o superiore a quella del nodo non riparabile, e inserisco in *codaSource* i nodi con label di uno inferiore a quella del nodo non riparabile.

Se *vuotiSink* non è vuota, provo a riparare tutti i nodi ivi contenuti: se riesco a ripararli tutti e *vuotiSource* è vuota, cerco un nodo di confine che sia risulti sia visitato sia valido, e, se riesce a raggiungere t, lo restituisco.

Altrimenti, al primo nodo non riparabile, mi fermo, salvo quel nodo, lo inserisco nuovamente in *vuotiSink* e riempio *codaSink* come segue :

- se il nodo è t, lo inserisco in *codaSink*
- altrimenti indico come non visitati i nodi con label pari o superiore a quella del nodo non riparabile, e inserisco in *codaSink* i nodi con label di uno inferiore a quella del nodo non riparabile.

Successivamente procedo con l'esplorazione del grafo, con uno degli algoritmi presentati nel capitolo 3.1.

Riparazione di un nodo

Con riparazione di un nodo *node* si indica cercare, tra i nodi a lui adiacenti, un generico nodo *substitute* che permetta di raggiungere il nodo da cui è stato esplorato, senza dover cercare nuovamente un percorso attraverso tutto il grafo. Se *node* è un nodo di confine e l'arco saturo lo collega a un nodo esplorato da source, devo cercare un *substitute* con le seguenti caratteristiche

- deve essere stato scoperto da s;
- deve essere valido;
- deve risultare visitato;
- l'arco che li collega deve avere, se entra in *node*, capacità residua positiva, invece, nel caso esca da *node*, flusso positivo;
- deve essere contenuto in LastNodesSourceSide;

Altrimenti *substitute* deve avere le seguenti caratteristiche:

- la label di *substitute* deve essere di uno inferiore a quella di *node*, quindi *substitute*.Label = *node*.Label -1;
- *substitute* deve essere della stessa parte di *node*;
- *substitute* deve essere valido;
- *substitute* deve risultare visitato;
- se *node* è stato scoperto da s, l'arco che collega *node* e *substitute* deve avere capacità positiva se entra in *node*, flusso positivo se entra in *substitute*;
- se *node* è stato scoperto da t, l'arco che collega *node* e *substitute* deve avere capacità positiva se esce in *node*, flusso positivo se ci entra.

Se trovo un nodo che soddisfa questi requisiti, allora posso aggiornare le informazioni di *node*, rendendolo valido e visitato, usando come nodo di indirizzamento *substitute*. Restituisco un booleano che mi indica se sono riuscito a riparare il nodo.

Per ulteriori dettagli si invita a guardare gli pseudo-codici 15, 16 e 20, presenti in appendice A.2.

3.4 Propagazione della malattia

3.4.1 Strutture dati

BiEdge

Rappresenta l'arco, è come quello presentato nel capitolo 3.2.1.

Node

Rappresenta il nodo, è rappresentato come descritto nel capitolo 3.2.1, ma con una proprietà dal funzionamento differente e due nuove proprietà:

- **Label**, rappresenta la distanza tra il nodo e, a seconda da chi l'ha scoperto, s o t;
- **SourceValid**, nuova proprietà, mi indica se il nodo precedessore è SourceValid o l'arco predecessore è saturo, quindi se è possibile raggiungere s.
- **SinkValid**, nuova proprietà, mi indica se il nodo successore è SinkValid o l'arco successore è saturo, quindi se è possibile raggiungere t.

Graph

Rappresenta il grafo, è come quello presentato nel capitolo 3.3.1

3.4.2 Descrizione

Data una rete (G, u, s, t), inizializzo a zero una variabile $fMax$ che mi terrà conto del valore del flusso inviato, e inizializzo le due pile $vuotiSource$, che conterrà i nodi non validi della parte di Source, e $vuotiSink$, che conterrà i nodi non validi della parte di Sink, inserendovi rispettivamente i nodi s e t.

Ciclicamente, cerco un cammino aumentante, se non lo trovo, restituisco il valore $fMax$ e termino l'esecuzione, altrimenti cerco il flusso inviabile attraverso quel cammino, svuoto le due pile $vuotiSource$ e $vuotiSink$, poi procedo a inviare il flusso attraverso il cammino indicato, se un arco si satura, vado a inserire il nodo che lo ha come PreviousEdge o come NextEdge nell'apposita pila, aggiornando $fMax$ col valore del flusso inviabile trovato.

Ricerca del cammino aumentante

Ricevo in input una rete (G, u, s, t), insieme a due pile: $vuotiSource$, contenente i nodi che non sono più SourceValid, e $vuotiSink$, contenente i nodi che non sono più SinkValid.

Se *vuotiSource* non è vuota, procedo a riparare i nodi contenuti al suo interno, nel caso non riesca a riparare un nodo, lo inserisco nella coda *malati*, indico che non sono riuscito a riparare tutti i nodi e salvo il primo nodo trovato che non è riparabile in una variabile *noCap*. Se sono riuscito a riparare tutti i nodi e la pila *vuotiSink* è vuota, allora cerco un nodo di confine che sia SourceValid sia SinkValid, e che possa raggiungere sia s, sia t, e lo restituisco, confermando che ho trovato un cammino aumentante.

Nel caso non sia riuscito a riparare tutti i nodi, procedo a propagare la malattia nella parte di Source per tutti i nodi contenuti in *malati*. Nel caso la propagazione della malattia mi restituisca un nodo di confine, che è SourceValid, e *vuotiSink* è vuota, indico che ho trovato un cammino aumentante e restituisco il nodo ottenuto dalla propagazione della malattia.

Se non ho nodi in *vuotiSink*, per ogni nodo di confine che sia SourceValid, e che ha il nodo successore, raggiungibile, che è in grado di raggiungere t, se è in grado di raggiungere s attraverso il percorso descritto dai PreviousNode, indico che ho trovato un cammino aumentante e lo restituisco.

Nel caso non sia abbia trovato un cammino aumentante, allora dovrò esplorare il grafo, se *codaSource* è ancora vuota, la inizializzo per l'esplorazione della parte Source, dipendentemente dal primo nodo che non sono riuscito a riparare *noCap*:

- se $noCap = s$, mi indica che è la prima iterazione, quindi procedo a inserire il nodo s in *codaSource*;
- se *noCap* è un nodo di confine, allora inserisco in *codaSource* tutti i nodi contenuti in LastNodesSourceSide;
- se la propagazione della malattia ha riparato almeno un nodo, inserisco in *codaSource* i nodi del grafo che hanno label pari al valore minimo di Label dei nodi riparati, e indico come non esplorati i nodi con label maggiore di quella trovata;
- altrimenti inserisco in *codaSource* i nodi che hanno label di uno precedente a quella di *noCap*, e indico come non visitati i nodi con label pari o superiore rispetto a quella di *noCap*.

Se *vuotiSink* non è vuota, procedo a riparare i nodi contenuti al suo interno, nel caso non riesca a riparare un nodo, lo inserisco in una coda *malati*, indico che non sono riuscito a riparare tutti i nodi e salvo il primo nodo trovato che non è riparabile in una variabile *noCap*.

Se sono riuscito a riparare tutti i nodi e avevo *vuotiSource* vuota o sono riuscito a riparare tutti i nodi contenuti in *vuotiSource*, allora cerco un nodo di confine che

sia SourceValid sia SinkValid, che possa raggiungere sia s, sia t, e lo restituisco, confermando che ho trovato un cammino aumentante.

Nel caso non sia riuscito a riparare tutti i nodi, procedo a propagare la malattia nella parte di Sink per tutti i nodi contenuti in *malati*. Nel caso la propagazione della malattia mi restituisca un nodo di confine, che è SinkValid, e la parte source è stata riparata, indico che ho trovato un cammino aumentante e restituisco il nodo ottenuto dalla propagazione della malattia.

Per ogni nodo di confine che sia SourceValid, e che ha il nodo precedessore, raggiungibile, che è in grado di raggiungere s, se è in grado di raggiungere t attraverso il percorso descritto dai NextNode, indico che ho trovato un cammino aumentante e lo restituisco.

Nel caso non abbia trovato un cammino aumentante, allora dovrò esplorare il grafo, se *codaSink* è ancora vuota, la inizializzo per l'esplorazione della parte Sink, dipendentemente dal primo nodo che non sono riuscito a riparare *noCap*:

- se $noCap = t$, mi indica che è la prima iterazione, quindi procedo a inserire il nodo t in *codaSink*;
- se la propagazione della malattia ha riparato almeno un nodo, inserisco in *codaSink* i nodi del grafo che hanno label pari al valore minimo di Label dei nodi riparati, e indico come non esplorati i nodi con label maggiore di quella trovata;
- altrimenti inserisco nella coda i nodi che hanno label di uno precedente a quella di *noCap*, e indico come non visitati i nodi con label pari o superiore rispetto a quella di *noCap*.

Successivamente procedo con l'esplorazione dei nodi, a seconda della strategia di esplorazione desiderata, presentate nel capitolo 3.1.

Propagazione della malattia

Ricevo in input il nodo non valido che deve propagare la malattia.

Inizializzo una variabile $min = +\infty$, e inserisco il nodo ricevuto in input in una coda *malati*, finché quella coda non sarà vuota, estraggo un nodo dalla coda e, se è stato esplorato dalla parte che sto analizzando, provo a ripararlo. Se non riesco a ripararlo, inserisco i suoi nodi adiacenti nella coda *malati*, altrimenti o, se il nodo non è di confine, salvo in *min* il valore minimo tra la sua label e *min*, se il nodo è di confine, lo restituisco.

Riparazione di un nodo

Cercare di riparare nodo *node* indica cercare, tra i nodi a lui adiacenti, un generico nodo *substitute* che permetta di raggiungere il nodo da cui è stato esplorato, senza dover cercare nuovamente un percorso attraverso tutto il grafo. Se *node* è un nodo di confine e l'arco saturo lo collega a un nodo esplorato da source, devo cercare un *substitute* con le seguenti caratteristiche

- deve essere stato scoperto da s;
- deve essere SourceValid;
- deve risultare visitato;
- l'arco che li collega deve avere, se entra in *node*, capacità residua positiva, invece, nel caso esca da *node*, flusso positivo;
- deve essere contenuto in LastNodesSourceSide;

Altrimenti *substitute* deve avere le seguenti caratteristiche:

- la label di *substitute* deve essere di uno inferiore a quella di *node*, quindi *substitute*.Label = *node*.Label -1;
- *substitute* deve essere della stessa parte di *node*;
- *substitute* deve essere valido rispetto alla parte esplorata;
- *substitute* deve risultare visitato;
- *substitute* deve avere non deve avere come PreviousNode o NextNode il nodo *node*
- se *node* è stato scoperto da s, l'arco che collega *node* e *substitute* deve avere capacità positiva se entra in *node*, flusso positivo se entra in *substitute*;
- se *node* è stato scoperto da t, l'arco che collega *node* e *substitute* deve avere capacità positiva se esce in *node*, flusso positivo se ci entra.

Se trovo un nodo che soddisfa questi requisiti, allora posso aggiornare le informazioni di *node*, rendendolo valido e visitato, usando come nodo di indirizzamento *substitute*. Restituisco un booleano che mi indica se sono riuscito a riparare il nodo.

Per ulteriori dettagli si invita alla visione degli pseudo-codici 15, 17, 18, 19 e 20, presenti in appendice A.6 .

3.5 Shortest Augmenting Path

3.5.1 Strutture dati

BiEdge

Rappresenta l'arco, è come descritto nel capitolo 3.2.1.

Node

Rappresenta il nodo, è caratterizzato dalle seguenti proprietà:

- **Name** rappresenta il nome dato al nodo;
- **Edges**, è la lista di archi a lui incidenti, nello pseudo-codice si indica con le funzioni δ, δ^+ per gli archi uscenti e δ^-, per gli archi entranti;
- **SourceDistance**, rappresenta la distanza tra il nodo e s, verrà usata anche la funzione d_s per rappresentarla;
- **SinkDistance**, rappresenta la distanza tra il nodo e t, verrà usata anche la funzione d_t per rappresentarla;
- **PreviousNode**, rappresenta il nodo da cui è stato esplorato, e quindi il nodo che posso usare per tornare al nodo sorgente s;
- **PreviousEdge**, rappresenta l'arco che collega il nodo al proprio PreviousNode;
- **NextNode**, rappresenta il nodo da cui è stato esplorato, e quindi il nodo che posso usare per tornare al nodo destinazione t;
- **NextEdge**, rappresenta l'arco che collega il nodo e NextNode.

I metodi di Node sono soltanto per cambiare valore alle variabili, con il metodo **Reset**, si intende cancellare i valori contenuti in NextEdge, NextNode, PreviousEdge, PreviousNode.

Graph

Rappresenta il grafo, è rappresentato attraverso un insieme che contiene tutti i nodi.

3.5.2 Descrizione

Data una rete (G, u, s, t), procedo a svolgere due Bfs, una a partire da s, l'altra a partire da t, per inizializzare la SourceDistance e SinkDistance di tutti i nodi del grafo, e procedo a inviare il flusso trovato da queste ricerche, salvando in flusso inviato in una variabile $fMax$, cancello le informazioni di indirizzamento dei nodi e, infine, inizializzo i due interi $fSource$ e $fSink$ entrambi a $+\infty$ e i due nodi $startSource$ e $StartSink$ rispettivamente ai nodi s e t.

Finché il flusso che ho inviato è positivo e $d_s(t)$ e $d_t(s)$ sono minori del numero di nodi del grafo, procedo come segue, altrimenti restituisco il valore di $fMax$.

Procedo a iniziare la ricerca di un cammino aumentante, dando in input i nodi di partenza, $startSource$ e $startSink$, e il valore del flusso di partenza, $fSource$ e $fSink$, oltre al puntatore di due code dove andrò a inserire i nodi esplorati, ricevo due valori interi $fSource$ e $fSink$ e due nodi $startSource$ e $startSink$, qui ho quattro possibili casi:

- $startSource = startSink$, quindi durante la ricerca la parte di source e la parte di sink si sono incontrate, invio il flusso dettato dai vari PreviousNode e NextNode del valore minimo tra $fSource$ e $fSink$, inizializzo il nodo $startSource$ a s e il nodo $startSink$ a t e i valori $fSource$ e $fSink$ entarmbi a $+\infty$, e cancello le informazioni di indirizzamento per ogni nodo contenuto nelle code per salvare i nodi esplorati;

- $startSink = s$, quindi la ricerca a partire di t ha raggiunto s, invio il flusso $fSink$ nel percorso descritto da s verso t, inizializzo $startSink$ a t e $fSink$ a $+\infty$,e cancello le informazioni di indirizzamento dai nodi contenuti nella coda dedicata a salvare i nodi esplorati dalla parte di sink;

- $startSource = t$, quindi la ricerca a partire di s ha raggiunto t, invio il flusso $fSource$ nel percorso descritto da t verso s, inizializzo $startSource$ a s e $fSource$ a $+\infty$, e cancello le informazioni di indirizzamento dai nodi contenuti nella coda dedicata a salvare i nodi esplorati dalla parte di source;

- se ricevo dei risultati non validi con i precedenti, termino l'esecuzione e restituisco il valore $fMax$.

Infine, aggiorno il valore di $fMax$

Ricerca di un cammino aumentante

Si usa una mutua ricorsione, quando eseguo un'operazione di advance a Source, chiamo la funzione per permettere di eseguire un advance a Sink e viceversa.

Ricevo in input una rete (G, u, s, t), i nodi $startSource$ e $startSink$, gli interi $sourceFlow$ e $sinkFlow$ e le code di nodi $codaSource$ e $codaSink$.

Se $startSource = startSink$, allora ritorno subito i valori ($sourceflow$, $sinkflow$, $startSource$, $startSink$); altrimenti controllo che

$$d_s(startSink) < \#V(G) \wedge d_t(startSource) < \#V(G)$$

altrimenti restituisco valori $(0, 0, null, null)$.

Cerco un nodo tale che mi possa permettere di avanzare, se lo trovo aggiorno il corrispettivo valore del flusso, aggiorno le informazioni di indirizzamento, inserisco il nodo trovato nella coda dei nodi esplorati della sua parte e, se sto esplorando con la parte di Sink e il nodo trovato è s, ritorno i valori $(sourceFlow, sinkFlow, startSource, s)$; se sto esplorando la parte di source e ho trovato t, ritorno i valori ($sourceFlow$, $sinkFlow$, t, $startSink$).

Se il nodo esplorato è già stato esplorato dalla parte opposta (quindi se ha un valore di indirizzamento), ritorno il valore ($sourceFlow$, $sinkFlow$, $node$, $node$), dove $node$ è il nodo trovato, altrimenti procedo a eseguire la ricerca dalla parte opposta.

Nel caso non abbia trovato alcun nodo che mi permettesse di eseguire un Advance, procedo con il retreat: per ogni arco che ha capacità positiva ed esce, nel caso si stia esplorando la parte di source, o che entra, nel caso stia esplorando la parte di sink, valuto la distanza minima, aggiorno la distanza del nodo di partenza ($startSource$ o $starSink$) con il valore minimo trovato + 1. Ripeto la ricerca (dalla stessa parte) con il nodo predecessore, se il nodo di partenza è diverso da s per l'esplorazione di source e diverso da t per l'esplorazione di sink, altrimenti con il nodo di partenza stesso.

Per ulteriori dettagli si invita a guardare lo pseudo-codice 21,22 e 23 ,presente in appendice A.7.

Capitolo 4

Risultati sperimentali

4.1 Creazione del grafo con valori pseudocasuali

Ricevo in input la cardinalità del grafo, che a cui aggiungerò un nodo finale, creo il grafo e il nodo sorgente s, lo inserisco nel grafo e in una lista. Creo i rimanenti nodi per arrivare alla cardinalità, li inserisco sia nel grafo sia nella lista, infine creo il nodo destinazione t, che inserirò opportunamente nel grafo e nella lista.

Per ogni nodo, escluso t, genero un valore pseudocasuale, compresso tra 1 e $(\#V(G) - i)\%(\#V(G)/10)$, dove con G si intende il grafo e con i la posizione nella coda: questo valore corrisponde a quanti archi uscenti può avere il nodo che sto analizzando.

Il nodo in cui l'arco entrerà è il nodo in posizione $i + x$, dove x è l'iterazione che sto facendo per la creazione e l'inserimento dell'arco.

Per ogni possibile arco, vado a generare pseudo-casualmente la capacità che dovrà avere, che sarà compressa tra 0 e 9999.

Se la capacità è diversa da 0, creo l'arco, con la capacità data, che collega i nodi x e $x + i$.

Per maggiore dettagli, si invita a vedere lo pseudo-codice 24 in appendice A.8.

4.2 Risultati ottenuti

I vari algoritmi sono stati implementati in C#, utilizzando .Net 6.0.100; la macchina su cui sono girati è stato un PC portatile ASUS X580VD, con processore Intel i7-7700HQ 2.8GHz, RAM 16 GB DDR4 2400 MHz, scheda grafica NVIDIA GeForce GTX 1050, inoltre i benchmark sono stati effettuati mentre il PC era in carica e con ventilazione esterna.

I grafi,creati come descritto nel capitolo 4.1, hanno 10001 nodi e in media 4553702 archi.

I tempi medi di esecuzione, in millisecondi, sono riportati tabella 1

Nella tabella 1 sono state riportate delle abbreviazioni, in seguito riportate:

- Label, si indica la strategie per l'esplorazione bidirezionale, che esplora i nodi con stessa label prima di proseguire con la parte opposta, presentata nel capitolo 3.1.1;
- NodeCount, si indica la strategia per l'esplorazione bidirezionale,che esplora un nodo alla volta per parte, presentata nel capitolo 3.1.2;
- NodePropagation, si indica la strategia per l'esplorazione bidirezionale,che esplora i nodi adiacenti al nodo esplorato per ogni parte, presentata nel capitolo 3.1.3.
- NoOpt, indica l'algoritmo di Edmonds-Karp senza alcuna ottimizzazione, presentato nei capitoli 2.1 e 3.2;
- LastLevelOpt, indica l'algoritmo con ottimizzazione negli ultimi livelli, presentato nei capitoli 2.2 e 3.3;
- SickPropagation, indica l'algoritmo con propagazione della malattia, presentato nei capitoli 2.3 e 3.4;
- SAP, indica l'algoritmo Shortest Augmenting Path, presentato nei capitoli 2.4 e 3.5, nella tabella si sono unite le tre strategie di esplorazione per rappresentare il suo tempo con l'algoritmo bidirezionale;

Algoritmi	Monodirezionali	Label	NodePropagation	NodeCount
NoOpt	7306.2	3542.95	3403.55	5961.65
LastLevelOpt	1135.15	1132.65	1305.35	2257.8
SickPropagation	187.95	624.65	718.4	950.3
SAP	1092055.25	183.5		

Tabella 1: Tempi medi per i vari algoritmi illustrati

Notiamo come Shortest Augmenting Path monodirezionale abbia le prestazioni peggiori, mentre le prestazioni di quello bidirezionali sono le migliori, ma questo potrebbe essere causato dalla creazione del grafo e dal generale basso numero di nodi entranti nel nodo destinazione.

La strategie di esplorazione migliore da usare per l'esplorazione bidirezionale, dai benchmark effettuati, è l'esplorazione dei nodi con pari label, mentre la peggiore è quella esplorando un nodo alla volta per parte.

Notiamo che, usando Edmonds-Karp senza alcuna ottimizzazione, i tempi migliori sono per gli algoritmi bidirezionali, sopratutto con NodePropagation, ma con l'ottimizzazione agli ultimi livelli, la differenza si assottiglia, ed è di poco migliore l'esplorazione tramite label rispetto a quella monodirezionale, mentre per la propagazione della malattia lo scarto che ha l'implementazione monodirezionale rispetto a quelle bidirezionali è notevole, questo potrebbe dover essere dovuto, per gli algoritmi bidirezionali, alla scelta di fermare l'esplorazione della propagazione della malattia ai nodi di confine e non procedere fino al nodo cercato, con la scelta effettuata si hanno molteplici nodi in cui cercare se è presente un percorso, inoltre è necessario un doppio controllo che sia possibili inviare flusso sia dalla parte esplorata dal nodo sorgente, sia dalla parte esplorata dal nodo destinazione.

Per vedere i benchmark fatti, si consiglia la visione delle tabelle presenti in appendice B

Conclusioni

Come abbiamo visto nel capitolo precedente, le prestazioni migliori sono state quelle dell'algoritmo Shortest Augmenting Path con implementazione bidirezionale, ma questo risultato potrebbe essere dato dalla creazione pseudocasuale del grafo. È stato inoltre illustrato uno delle possibili cause del fatto che l'implementazione monodirezionale dell'algoritmo di Edmonds-Karp con la propagazione della malattia termini in tempo minore rispetto alle implementazioni bidirezionali.

Sarebbe interessante, oltre a rendere gli algoritmi paralleli, esplorando i nodi esplorati a partire dal nodo sorgente e i nodi esplorati a partire dal nodo destinazione parallelamente, cambiare idea dietro alla propagazione della malattia bidirezionale, invece di bloccarsi a metà del percorso, essere disposti a sconfinare nella parte esplorata dalla parte opposta, permettendo così un controllo del possibile percorso più veloce.

Dato che i grafi con cui sono stati effettuati i benchmark vengono da uno stesso algoritmo, sarebbe interessanti migliorare i grafi usati come benchmark, usando, ad esempio quelli presenti nel dataset in Ref[3], nel caso trasformando i dati grazie ai programmi presentati in Ref[4].

Ringraziamenti

Le persone che mi hanno accompagnato in questi ultimi anni durante il periodo universitario sono molte. Le prime persone che vorrei ringraziare sono i miei genitori, che mi hanno permesso di trasferirmi a Milano e mi hanno mantenuto, permettendomi di fare ciò che piace.

Ringrazio tutti i professori che ho incontrato nel mio percorso, per avermi passato fatto appassionare alle loro materie, un particolare ringraziamento il professor Giovanni Righini che mi ha permesso di fare questa tesi, farmi approfondire questo argomento così interessante e utile, nella speranza, durante la magistrale, di poterlo approfondire molto di più.

Ringrazio i miei parenti tutti, che mi hanno sempre aiutato al meglio delle loro possibilità, preoccupandosi per me, quasi come fossi un loro figlio. Ringrazio la famiglia Fedeli, che mi è stata sempre vicino.

Ringrazio tutti i coinquilini che in questi quattro anni ho avuto: Andrea Aime, chiamato per ancora non si sa quale motivo muschio, che mi ha accompagnato durante il mio secondo anno e quest'anno mio compagno di tesi, nonostante la sua sia palesemente migliore dalla mia, Nello, che col tuo fon mi hai aperto gli occhi, dimostrandomi che un matematico può essere in parte informatico, e farmi chiedere se è possibile fare il contrario, Jack, te che hai deciso di voler vivere l'università oltre il massimo, ti ho visto che non bastavi da solo, e non ho potuto fare niente per poterti riprendere, Fratta, primo fisico con cui vivo, prima volta che capisco cosa vuol dire, per gli altri, aver a che fare con un informatico come me, John, spero che troverai presto la strada che fa per te, così come, credo, di averla trovata io, Charo, ti ringrazio di non aver sopportato un coinquilino così nottambulo, nonostante la notte la passi sul PC, Paco, con la tua allegria e voglia ci facevi stare bene, Kid, ricordati che Alessandria non è la capitale D'Italia, e non tutti siamo interessati a quante ne prendi, Berto, che mi hai insegnato un sacco di cose sulle vacche, e un giorno mi spiegherai a cosa servono a un informatico tutte queste vaccate, Shu, caro compagno giapponese che bevevi un bicchiere di vino ed eri già ubriaco, non sai quanto manchi, Genna, con il tuo incredibile cuore e la tua incredibilità capacità di fare tutto quello che volevi, Zibo, grazie per avermi fatto capire come coltivare le mie passioni anche in appartamento: divertendosi, Passe, nonostante la tua pazzia dilagante, il silenzio dovuta alla

tua mancanza si è fatta sentire da quando ho rimesso piede in appartamento, Lollo e Michelotti, che mi avete mostrato il mondo della letteratura, e avermi fatto capire che l'unica adatta a me è quella scientifica, infine come non posso ringraziare i due miei due coinquilini informatici: Water, colui che, solo con la propria passione, mi ha fatto capire cosa vuol dire essere un informatico, e Siri, che mi ha sempre accompagnato nel mio percorso, a cui ho potuto chiedere un aiuto, che poi mi fosse dato era un altro discorso.

Ringrazio i miei "coinquilini" che mi hanno ospitato da loro e offerto la loro compagnia durante il primo lockdown: Bobby, con la fuga prima del dovuto e con lei le nostre notti di studio insieme mancate, cippolo, con la tua sfrenata passione per la musica, spero di poterti rivedere prima o poi, Frienz, con il tuo studio, i tuoi aiuti e la tua voglia di fare.

Ringrazio chi ha fatto parte dei miei momenti di riposo, che sia stato per giocare di ruolo o con qualche videogioco, e, nonostante gli screzi, ci siamo divertiti.

Ringrazio invece chi, oltre ai momenti di riposo, ha condiviso con me i momenti di studio e di lavoro: Mandiz, la prima persona che ho incontrato in università, speravo che tu fossi qui al mio fianco pronto a discutere la tua tesi appena dopo di me, Filippo Fantinato, con la tua voglia e le tue battute, spero di rivederti presto, Orfei, spero che capirai presto cosa vuoi fare, se finire l'università, continuare a studiare o procedere del tuo lavoro, Manu, te, con tutte le tue particolarità, che nonostante il tuo essere così troppo pieno di te, riesci comunque a essere un buon amico, Tricella, un gigante dal cuore d'oro, nonostante il tuo essere un po' polemico, sei sempre pronto ad aiutare chi ha bisogno di una mano, Ladisa, nonostante il poco studio fatto con te, spero che anche tu riesca a capire cosa vuoi fare e tu riesca a farlo, Dibbi, spero che, con la tua passione per la musica, riesca a raggiungere i tuoi obiettivi, Geologo, dopo aver tentato geologia, spero che informatica sia la via giusta per te, ultimo, ma non meno importante, Scutta, spero di poter tornare a studiare con te, o meglio, che tu torni a studiare con me.

Infine voglio ringraziare tutti coloro che mi hanno permesso di essere qui, in piedi, che essi siano stati medici, infermieri, fisioterapisti o volontari, è grazie anche grazie a voi che sono riuscito ad arrivare fino a qui.

Bibliografia

[1] Korte, Bernhard, and Springer Publishing Company. *Ottimizzazione combinatoria: teoria e algoritmi*. Milano: Springer, 2011. Print.

[2] Ahuja, Ravindra K., and Prentice-Hall. *Network Flows: Theory, Algorithms and Applications*. Englewood Cliffs (N.J.): Prentice Hall, 1993. Print.

[3] Patrick M. Jensen, Niels Jeppesen, Anders B. Dahl, & Vedrana A. Dahl. (2021). *Min-Cut/Max-Flow Problem Instances for Benchmarking* [Data set]. Zenodo. https://doi.org/10.5281/zenodo.4905882

[4] *Review of Serial and Parallel Min-Cut/Max-Flow Algorithms for Computer Vision*, Patrick M. Jensen, Niels Jeppesen, Anders B. Dahl, Vedrana A. Dahl, 2022 (Under review).

Appendice A

Pseudo-codice

A.1 Pseudo-codice algoritmo monodirezionale di ricerca del flusso massimo senza alcuna ottimizzazione

Algorithm 1 Inizializzazione variabili e invio del flusso

Require: Rete (G, u, s, t)
Ensure: quantità del flusso massima inviata, grafo dei residui aggiornato

```
1:  fMax ← 0
2:  while true do
3:      cerca un cammino aumentante
4:      if t.FlussoPassante = 0 then
5:          return fMax
6:      end if
7:      fMax ← fMax + t.FlussoPassante
8:      node ← t
9:      while node ≠ s do
10:         node.Previous.AddFlow(t.FlussoPassante, node)
11:         node ← node.PreviousNode
12:     end while
13: end while
```

Algorithm 2 Ricerca del cammino aumentante

Require: rete (G, u, s, t)
Ensure: aggiorno informazioni

```
1: Reset()          ▷ cancello FlussoPassante e PreviousNode per ogni nodo,tranne s
2: coda ← coda di nodi,contenente s
3: while ¬coda.isEmpty do
4:     element ← coda.Dequeue()
5:     for all edge ∈ δ(element)|edge.NextNode.FlussoPassante = 0 ∧
   ((u_f(edge) > 0 ∧ ¬edge is ReversedMonoEdge) ∨ (f(edge) > 0 ∧
   edge is ReversedMonoEdge)) do
6:         n ← edge.NextNode
7:         n.update(element, edge)
8:         if n = t then
9:             return
10:        else
11:            coda.enqueue(n)
12:        end if
13:    end for
14: end while
```

A.2 Pseudo-codice di ricerca del flusso massimo monodirezionale con ottimizzazione sugli ultimi livelli e propagazione della malattia

Algorithm 3 Inizializzazione variabili e invio del flusso

Require: rete (G, u, s, t)
Ensure: quantità di flusso massima inviabile, grafo dei residui aggiornato

```
1:  fMax ← 0
2:  malati ← pila vuota di nodi
3:  while true do
4:      ricerca di un cammino aumentante(G, malati)        ▷ vedesi algoritmi 4 o 6
5:      f ← +∞
6:      node ← t
7:      while node ≠ s do
8:          if node.PreviousEdge.Reversed then
9:              f ← min(f(node.PreviousEdge), f)
10:         else
11:             f ← min(u_f(node.PreviousEdge), f)
12:         end if
13:         node ← node.PreviousNode
14:     end while
15:     if f = 0 then
16:         break
17:     end if
18:     node ← t
19:     malati.Clear()                          ▷ li faccio diventare nuovamente vuoti
20:     while node ≠ s do
21:         if mom.PreviousEdge.AddFlow(f) then
22:             malati.Push(node)
23:         end if
24:         mom ← mom.PreviousNode
25:     end while
26:     fMax ← fMax + f
27: end while
28: return fMax
```

Algorithm 4 Ricerca del cammino aumentante con ottimizzazione sugli ultimi livelli

Require: rete (G, u, s, t), pila di nodi *malati*
Ensure: aggiornamento del grafo dei residui

```
1: coda ← coda vuota di nodi
2: if malati.isEmpty then
3:     coda.enqueue(s)
4:     for all n inV(G) \ s do
5:         n.Reset()
6:     end for
7: else
8:     repaired ← true
9:     while ¬malati.isEmpty do
10:        n ← malati.Pop()
11:        if ¬ Repair(n) then                    ▷ vedesi l'algoritmo 5
12:            repaired ← false
13:            break
14:        end if
15:    end while
16:    if repaired then
17:        return
18:    end if
19:    for all node ∈ V(G)|n.Label − 1 = node.Label do
20:        coda.enqueue(node)
21:    end for
22:    for all node ∈ V(G)|node.Label ≥ n.Label do
23:        node.Reset()
24:    end for
25: end if
26: while ¬coda.isEmpty do
27:    element ← coda.dequeue()
28:    if ¬element.Valid then
29:        continue
30:    end if
```

31: **for all** $edge \in \delta^+(element) | u_f(edge) > 0 \wedge (\neg edge.NextNode.Visited \vee \neg edge.NextNode.Valid$ **do**
32: $n \leftarrow edge.NextNode$
33: n.update$(p, edge)$ ▷ aggiornamento dei dati di label,visited e indirizzamento
34: **if** $n = t$ **then**
35: **return**
36: **end if**
37: $coda$.enqueue(n)
38: **end for**
39: **for all** $edge \in \delta^-(element) | f(edge) > 0 \wedge (\neg edge.PreviousNode.Visited \vee \neg edge.PreviousNode.Valid$ **do**
40: $p \leftarrow edge.PreviousNode$
41: p.update$(n, edge)$
42: **if** $p = t$ **then**
43: **return**
44: **end if**
45: $coda$.enqueue(p)
46: **end for**
47: **end while**

Algorithm 5 Riparazione di un nodo

Require: nodo da riparare *node*
Ensure: true se riesco a riparare il nodo, false altrimenti

1: **for all** $edge \in \delta^+(node) | u_f(edge) > 0 \wedge edge.PreviousNode.Label = node.Label - 1 \wedge edge.PreviousNode.Valid \wedge edge.PreviousNode.Visited$ **do**
2: $node$.update$(edge.PreviousNode, edge)$
3: **true**
4: **end for**
5: **for all** $edge \in \delta^-(node) | f(edge) > 0 \wedge edge.NextNode.Label = node.Label - 1 \wedge edge.NextNode.Valid \wedge edge.NextNode.Visited$ **do**
6: $node$.update$(edge.NextNode, edge)$
7: **return** true
8: **end for**
9: InvalidNode$(node)$
10: **return** false

Algorithm 6 Ricerca del cammino aumentante con propagazione della malattia

Require: rete (G, u, s, t), pila di nodi *noCaps*
Ensure: grafo dei residui aggiornato

```
1:  coda ← coda vuota di nodi
2:  malati ← coda di nodi vuota
3:  fromSource ← false
4:  if noCaps.isEmpty then
5:      coda.Enqueue(s)
6:      fromSource ← true
7:  else
8:      repaired ← true
9:      while ¬noCaps.isEmpty do
10:         noCap ← noCapsPop()
11:         if ¬Repair(noCap) then                ▷ vedesi l'algoritmo 5
12:             malati.enqueue(noCap)
13:             repaired ← false
14:         end if
15:     end while
16:     if repaired then
17:         return
18:     end if
19:     firstNoCap ← null
20:     while ¬malati.isEmpty do
21:         noCap ← malati.dequeue()
22:         SickPropagation(noCap, coda)          ▷ vedesi l'algoritmo 7
23:         if firstNoCap = null then
24:             firstNoCap ← noCap
25:         end if
26:     end while
27:     if t.Valid then
28:         return
29:     end if
```

30: **for all** $n \in V(G) | n.Label \geq firstNoCap.Label$ **do**
31: n.Reset()
32: **end for**
33: **if** $coda.isEmpty$ **then**
34: **for all** $n \in V(G) | n.Label = firstNoCap.Label - 1$ **do**
35: $coda$.enqueue(n)
36: **end for**
37: **end if**
38: **end if**
39: **while** $\neg coda.isEmpty$ **do**
40: $element \leftarrow coda$.dequeue
41: **if** $\neg element.Valid \vee \neg element.Visited$ **then**
42: **continue**
43: **end if**
44: **for all** $edge \in \delta^{+}(element) | \neg edge.NextNode.Visited \wedge u_f(edge) > 0 \wedge (edge.NextNode.Label > element.Label \vee fromSource \vee \neg edge.NextNode.Valid)$ **do**
45: $n \leftarrow edge.NextNode$
46: n.update($element, edge$)
47: **if** $n = t$ **then**
48: **return**
49: **else**
50: $coda$.enqueue(n)
51: **end if**
52: **end for**
53: **for all** $edge \in \delta^{-}(element) | \neg edge.PreviousNode.Visited \wedge f(edge) > 0 \wedge (edge.PreviousNode.Label > element.Label \vee fromSource \vee \neg edge.PreviousNode.Valid)$ **do**
54:
55: $p \leftarrow edge.PreviousNode$
56: p.update($element, edge$)
57: **if** $p = t$ **then**
58: **return**
59: **else**
60: $coda$.enqueue(p)
61: **end if**
62: **end for**
63: **end while**

Algorithm 7 Propagazione della malattia

Require: nodo da riparare *node*, coda dei validi da esplorare *coda*
Ensure: grafo dei residui aggiornato

```
1: malati ← coda di nodi contenente node
2: while ¬malati.isEmpty do
3:     m ← malati.dequeue()
4:     if ¬Repair(m) then                          ▷ vedesi l'algoritmo 5
5:         for all edge ∈ δ(m) do
6:             p ← edge.PreviousNode
7:             n ← edge.NextNode
8:             if p = m ∧ m = n.PreviousNode then
9:                 malati.enqueue(n)
10:            else if n = m ∧ m = p.PreviousNode then
11:                malati.enqueue(p)
12:            end if
13:        end for
14:    else if m = t then
15:        return
16:    else
17:        coda.enqueue(m)
18:    end if
19: end while
```

A.3 Pseudocodice dell'algoritmo Shortest Augmenting Path monodirezionale

Algorithm 8 Inizializzazione delle variabili e operazione Augment

Require: Rete (G, u, s, t)
Ensure: quantità del flusso massima inviata, grafo dei residui aggiornato

```
1: fMax ←Bfs(t)      ▷ Semplice bfs,da t, analizza tutti i nodi, aggiornando i nodi, inoltre indica anche un cammino da un generico nodo verso t, s compreso
2: SendFlow(fMax, s)   ▷ invio il flusso fMax negli archi indicati da previousEdge del nodo s
3: for all n ∈ V(G) do
4:     n.Reset()                          ▷ cancello previousEdge e PreviousNode
5:     esplorati ← coda di nodi vuota
6: end for
7: while d(s) < #V(G) do                  ▷ d è la funzione distanza
8:     f gets CamminoAumentante (G, esplorati)
9:     fMax ← fMax + f
10:    if f = 0 then
11:        break
12:    end if
13:    SendFlow(f, t)
14:    while ¬esplorati.isEmpty do
15:        esplorati.dequeue().Reset()
16:    end while
17: end while
18: return fMax
```

Algorithm 9 Ricerca del cammino aumentante

Require: rete (G, u, s, t), coda *esplorati*
Ensure: valore del flusso inviabile

```
1: pila ← pila della coppia (nodo,intero)
2: pila.Push(s, −+∞)
3: while ¬pila.isEmpty do
4:     advanced ← false
5:     (start, f) ← pila.Pop()
6:     if d(start) < #V(G) then
7:         for all edge ∈ δ+(start)|d(edge.NextNode) = d(start) − 1 ∧ u_f(edge) > 0
   do
8:             f ← min(f, u_f(edge))
9:             n ← edge.NextNode
10:            n.SetPrevious(e)      ▷ aggiorna dati di PreviousNode e PreviousEdge
11:            esplorati.enqueue(n)
12:            if n = t then
13:                return f
14:            end if
15:            pila.Push((n, f))
16:            advanced ← true
17:            break
18:        end for
19:        if ¬advanced then
20:            min ← +∞
21:            for all edge ∈ δ+(start)|u_f(edge) > 0 do
22:                min ← min(min, d(edge.NextNode))
23:            end for
24:            d(start) ← min
25:            if start = s then
26:                start pila.Push((start, f))
27:            else
28:                start pila.Push((start.PreviousNode, f))
29:            end if
30:        end if
31:    else
32:        return 0
33:    end if
34: end while
35: return 0
```

A.4 Strategie per esplorazione bidirezionale

Algorithm 10 Pseudo-codice dell'esplorazione bidirezionale con esplorazione di nodi con stessa Label

Require: rete (G, u, s, t), code di nodi *codaSource* e *codaSink*, booleani sourceRepaired e sinkRepaired

Ensure: nodo di confine contenente le informazioni per raggiungere s e t

```
1:  needSink ← false
2:  codaBuffer ← coda di nodi vuota
3:  do
4:      if needSink then
5:          for all n ∈ V(G)|¬n.SourceSide ∧ n ≠ t do
6:              n.Reset()
7:          end for
8:          codaSink.enqueue(t)
9:          needSink ← false
10:     end if
11:     while ¬codaSource.isEmpty ∨ ¬codaSink.isEmpty  do
12:         while ¬codaSource.isEmpty do
13:             element ← codaSource.dequeue()
14:             if ¬element.sourceSide ∧ ¬element.Visited ∧ ¬element.Valid then
15:                 continue
16:             end if
17:             for all edge ∈ δ(element) do
18:                 p ← edge.previousNode
19:                 n ← edge.nextNode
20:                 if element = p ∧ u_f(edge) > 0 then
21:                     if n.Visited ∧ ¬n.sourceSide then
22:                         n.updatePath(p, edge)          ▷ aggiorno le informazioni di
    indirizzamento e di validità
23:                         graph.AddLast(n)
24:                         edge.reversed ← false
25:                         return n
26:                     else if ¬n.Visited ∧ n.sourceSide then
27:                         n.update(p, edge)                                    ▷
    aggiorno le informazioni di indirizzamento,label, validità, parte di esplorazione e
    indico il nodo come esplorato
```

28: $edge.reversed \leftarrow$ false
29: $codaBuffer$.enqueue(n)
30: **else if** $\neg n.sourceSide \wedge \neg n.Visited \wedge$ non è possibile esplorare la parte di Sink **then**
31: $needSink \leftarrow$ true
32: **end if**
33: **else if** $element = n \wedge f(edge) > 0$ **then**
34: **if** $p.Visited \wedge \neg p.sourceSide$ **then**
35: p.updatePath($n, edge$)
36: $graph$.AddLast(p)
37: $edge.reversed \leftarrow$ false
38: **return** p
39: **else if** $p.sourceSide \wedge \neg p.Visited$ **then**
40: p.update($n, edge$)
41: $edge.reversed \leftarrow$true
42: $codaBuffer$.enqueue(p)
43: **else if** $\neg p.sourceSide \wedge \neg p.Visited \wedge$ non è possibile esplorare la parte di Sink **then**
44: $needSink \leftarrow$ true
45: **end if**
46: **end if**
47: **end for**
48: **end while**
49: $mom \leftarrow codaBuffer$ ▷ scambio di puntatori tra codaBuffer e codaSource
50: $codaSource \leftarrow codaBuffer$
51: $codaBuffer \leftarrow mom$
52: **while** $\neg codaSink.isEmpty$ **do**
53: $element \leftarrow codaSink$.dequeue()
54: **if** $element.sourceSide \vee \neg element.Visited \vee \neg element.Valid$ **then**
55: **continue**
56: **end if**
57: **for all** $edge \in \delta(element)$ **do**
58: $p \leftarrow edge$.previousNode
59: $n \leftarrow edge$.nextNode

```
60:                 if element = n ∧ u_f(edge) > 0 then
61:                     if p.Visited then
62:                         if ¬p.sourceSide then
63:                             continue
64:                         else
65:                             n.updatePath(p, edge)
66:                             graph.AddLast(n)
67:                             edge.reversed ←false
68:                             return n
69:                         end if
70:                     end if
71:                     p.update(n, edge)
72:                     edge.reversed ←false
73:                     codaBuffer.enqueue(p)
74:                 end if
75:                 if element = p ∧ f(edge) > 0 then
76:                     if n.Visited then
77:                         if ¬n.sourceSide then
78:                             continue
79:                         else
80:                             p.updatePath(n, edge)
81:                             graph.AddLast(p)
82:                             edge.reversed ←true
83:                             return p
84:                         end if
85:                     end if
86:                     n.update(p, edge)
87:                     edge.reversed ←true
88:                     codaBuffer.enqueue(n)
89:                 end if
90:             end for
91:         end while
92:         mom ← codaBuffer    ▷ scambio di puntatori tra codaBuffer e codaSink
93:         codaSink ← codaBuffer
94:         codaBuffer ← mom
95:     end while
96: while needSink
97: return null
```

Algorithm 11 Pseudo-codice dell'esplorazione bidirezionale esplorando un nodo per ogni parte

Require: rete (G, u, s, t), code di nodi *codaSource* e *codaSink*, booleani SourceRepaired e SinkRepaired

Ensure: nodo di confine contenente le informazioni per raggiungere s e t

1: $needSink \leftarrow$ false
2: **do**
3: **if** $needSink$ **then**
4: **for all** $n \in V(G) | \neg n.SourceSide \wedge n \neq t$ **do**
5: n.Reset()
6: **end for**
7: $codaSink$.enqueue(t)
8: $needSink \leftarrow$ false
9: **end if**
10: **while** $\neg codaSink.isEmpty \vee \neg codaSource.isEmpty$ **do**
11: **if** $\neg codaSource.isEmpty \wedge (codaEdgeSource.isEmpty \vee (codaSink.isEmpty \wedge codaEdgeSink.isEmpty \wedge \neg sinkRepaired))$ **then**
12: $elementSource \leftarrow codaSource$.dequeue()
13: **if** $\neg elementSource.SourceSide \vee \neg elementSource.Valid \vee \neg elementSource.Visited$ **then**
14: **continue**
15: **end if**
16: **for all** $edge \in \delta^+(elementSource) | u_f(edge) > 0 \wedge (\neg edge.NextNode.Visited \vee \neg edge.NextNode.SourceSide)$ **do**
17: $codaEdgeSource$.enqueue($edge$)
18: **end for**
19: **for all** $edge \in \delta^-(elementSource) | f(edge) > 0 \wedge (\neg edge.PreviousNode.Visited \vee \neg edge.PreviousNode.SourceSide)$ **do**
20: $codaEdgeSource$.enqueue($edge$)
21: **end for**
22: **end if**
23: **if** $(\neg codaSink.isEmpty \wedge (codaEdgeSink.isEmpty \vee (codaSource.isEmpty \wedge codaEdgeSource.isEmpty \wedge \neg sourceRepaired))$ **then**
24: $elementSink \leftarrow codaSink.dequeue()$
25: **if** $elementSink.SourceSide \vee \neg elementSink.Valid \vee \neg elementSink.Visited$ **then**
26: **continue**
27: **end if**

28: **for all** $edge \in \delta^-(elementSink) | u_f(edge) > 0 \wedge (\neg edge.PreviousNode.Visited \vee edge.PreviousNode.SourceSide))$ **do**
29: $codaEdgeSink$.enqueue($edge$)
30: **end for**
31: **for all** $edge \in \delta^+(elementSink) | f(edge) > 0 \wedge (\neg edge.NextNode.Visited \vee edge.NextNode.SourceSide)$ **do**
32: $codaEdgeSink$.enqueue($edge$)
33: **end for**
34: **end if**
35: **while** $(\neg codaEdgeSource.isEmpty \vee sourceRepaired) \wedge (\neg codaEdgeSink.isEmpty \wedge sinkRepaired$ **do** ▷ con sourceRepaired e sinkRepaired si intende, più largamente, se è possibile esplorare quella parte di grafo
36: **if** $\neg codaEdgeSource.isEmpty$ **then**
37: $sourceEdge \leftarrow codaEdgeSource$.dequeue()
38: $p \leftarrow sourceEdge.previousNode$
39: $n \leftarrow sourceEdge.nextNode$
40: **if** $elementSource = p \wedge u_f(sourceEdge) > 0$ **then**
41: **if** $n.visited \wedge \neg n.sourceSide \wedge n.valid$ **then**
42: n.updatePath($p, sourceEdge$)
43: $sourceEdge$.Reversed$\leftarrow$ false
44: **return** n
45: **else if** $n.SourceSide \wedge (\neg n.Visited \vee \neg n.Valid)$ **then**
46: n.update($p, sourceEdge$)
47: $sourceEdge$.Reversed$\leftarrow$ false
48: $codaSource$.enqueue(n)
49: **else if** $\neg n.SourceSide \wedge (\neg n.valid \vee \neg n.visited) \wedge codaSink.isEmpty \wedge codaEdgeSink.isEmpty \wedge sinkRepaired$ **then**
50: $needSink \leftarrow$ true
51: **end if**
52: **else if** $elementSource = n \wedge f(sourceEdge) > 0$ **then**
53: **if** $p.Visited \wedge \neg p.SourceSide \wedge p.Valid$ **then**
54: p.updatePath($n, sourceEdge$)
55: $sourceEdge$.reversed $\leftarrow$ false
56: **return** p
57: **else if** $p.SourceSide \wedge (\neg p.Visited \vee p.Valid)$ **then**
58: p.update($n, sourceEdge$)
59: $sourceEdge$.reversed $\leftarrow$ false
60: $codaSource$.enqueue(p)

```
61:                 else if ¬p.SourceSide ∧ (¬p.Valid ∨ ¬p.Visited) ∧
    codaSink.isEmpty ∧ codaEdgeSink.isEmpty ∧ sinkRepaired then
62:                     needSink ← true
63:                 end if
64:             end if
65:         end if
66:         if ¬codaEdgeSink.isEmpty then
67:             edgeSink ← codaEdgeSink.dequeue()
68:             p ← edgeSink.previousNode
69:             n ← edgeSink.nextNode
70:             if elementSink = n ∧ u_f(edgeSink) > 0 then
71:                 if p.visited ∧ p.Valid then
72:                     if ¬p.sourceSide then
73:                         continue
74:                     else
75:                         n.updatePath(p, edgeSink)
76:                         edgeSink.reversed ←false
77:                         return n
78:                     end if
79:                 end if
80:                 p.update(n, edgeSource)
81:                 edgeSink.reversed ← false
82:                 codaSink.enqueue(p)
83:             else if elementSink = p ∧ f(elementSink) > 0 then
84:                 if n.visited ∧ n.Visited then
85:                     if ¬n.sourceSide then
86:                         continue
87:                     else
88:                         p.update(n, edgeSink)
89:                         return p
90:                     end if
91:                 end if
92:                 n.update(p, edgeSink)
93:                 edgeSink.reversed ← true
94:                 codaSink.enqueue(n)
95:             end if
96:         end if
97:     end while
98: end while
99: while needSink
```

Algorithm 12 Pseudo-codice dell'esplorazione bidirezionale con propagazione dei nodi

Require: rete (G, u, s, t), code di nodi *codaSource* e *codaSink*, booleani SourceRepaired e SinkRepaired

Ensure: nodo di confine contenente le informazioni per raggiungere s e t

```
1: needSink ← false
2: do
3:     if needSink then
4:         for all n ∈ V(G)|¬n.SourceSide ∧ n ≠ t do
5:             n.Reset()
6:         end for
7:         codaSink.enqueue(t)                    ▷ t è il nodo destinazione del grafo
8:         needSink ← false
9:     end if
10:    while ¬codaSource.isEmpty ∨ ¬codaSink.isEmpty  do
11:        if ¬codaSource.isEmpty then
12:            element ← codaSource.dequeue()
13:            if ¬element.sourceSide ∧ ¬element.Visited ∧ ¬element.Valid then
14:                continue
15:            end if
16:            for all edge ∈ δ(element) do
17:                p ← edge.previousNode
18:                n ← edge.nextNode
19:                if element = p ∧ u_f(edge) > 0 then
20:                    if n.Visited ∧ ¬n.sourceSide then
21:                        n.updatePath(p, edge)        ▷ aggiorno le informazioni di
    indirizzamento e di validità
22:                        graph.AddLast(n)
23:                        edge.reversed ← false
24:                        return n
25:                    else if ¬n.Visited ∧ n.sourceSide then
26:                        n.update(p, edge)                ▷
    aggiorno le informazioni di indirizzamento,label, validità, parte di esplorazione e
    indico il nodo come esplorato
27:                        edge.reversed ← false
28:                        codaSource.enqueue(n)
```

```
29:                 else if ¬n.sourceSide ∧ ¬n.Visited∧ non è possibile esplorare
    la parte di Sink then
30:                     needSink ← true
31:                 end if
32:             else if element = n ∧ f(edge) > 0 then
33:                 if p.Visited ∧ ¬p.sourceSide then
34:                     p.updatePath(n, edge)
35:                     graph.AddLast(p)
36:                     edge.reversed ← false
37:                     return p
38:                 else if p.sourceSide ∧ ¬p.Visited then
39:                     p.update(n, edge)
40:                     edge.reversed ←true
41:                     codaSource.enqueue(p)
42:                 else if ¬p.sourceSide ∧ ¬p.Visited∧ non è possibile esplorare la
    parte di Sink then
43:                     needSink ← true
44:                 end if
45:             end if
46:         end for
47:     end if
48:     if ¬codaSink.isEmpty then
49:         element ← codaSink.dequeue()
50:         if element.sourceSide ∨ ¬element.Visited ∨ ¬element.Valid then
51:             continue
52:         end if
53:         for all edge ∈ δ(element) do
54:             p ← edge.previousNode
55:             n ← edge.nextNode
56:             if element = n ∧ u_f(edge) > 0 then
57:                 if p.Visited then
58:                     if ¬p.sourceSide then
59:                         continue
60:                     else
61:                         n.updatePath(p, edge)
62:                         graph.AddLast(n)
63:                         edge.reversed ←false
64:                         return n
65:                     end if
66:                 end if
```

```
67:                     p.update(n, edge)
68:                     edge.reversed ←false
69:                     codaSink.enqueue(p)
70:                 end if
71:                 if element = p ∧ f(edge) > 0 then
72:                     if n.Visited then
73:                         if ¬n.sourceSide then
74:                             continue
75:                         else
76:                             p.updatePath(n, edge)
77:                             graph.AddLast(p)
78:                             edge.reversed ←true
79:                             return p
80:                         end if
81:                     end if
82:                     n.update(p, edge)
83:                     edge.reversed ←true
84:                     codaSink.enqueue(n)
85:                 end if
86:             end for
87:         end if
88:     end while
89: while needSink
```

A.5 Pseudo-codice algoritmo bidirezionale di ricerca del flusso massimo senza alcuna ottimizzazione

Algorithm 13 Invio del flusso e inizializzazione delle variabile per problema del flusso massimo bidirezionale senza nessuna ottimizzazione

Require: rete (G, u, s, t)
Ensure: quantità di flusso inviata, grafo dei residui aggiornato

1: $resetSource \leftarrow$ true
2: $resetSink \leftarrow$ true
3: $fMax \leftarrow 0$
4: **while** true **do**
5: $n \leftarrow$ ricerca del cammino aumentante $(G, resetSource, resetSink)$ ▷ si invita a guardare l'algoritmo 14
6: **if** $n =$ null **then**
7: **break**
8: **end if**
9: $flussoInviabile \leftarrow +\infty$
10: $mom \leftarrow n$
11: **while** $mom \neq s$ **do**
12: **if** $mom.PreviousEdge.Reversed$ **then**
13: $flussoInviabile \leftarrow \min(flussoInviabile, f(mom.PreviosEdge))$
14: **else**
15: $flussoInviabile \leftarrow \min(flussoInviabile, u_f(mom.PreviosEdge))$
16: **end if**
17: $mom \leftarrow mom.PreviosNode$
18: **end while**
19: $mom \leftarrow n$

```
20:     while mom ≠ t do
21:         if mom.NextEdge.Reversed then
22:             flussoInviabile ← min(flussoInviabile, f(mom.NextEdge))
23:         else
24:             flussoInviabile ← min(flussoInviabile, u_f(mom.NextEdge))
25:         end if
26:         mom ← mom.NextEdge
27:     end while
28:     resetSource ← false
29:     resetSink ← false
30:     mom ← n
31:     while mom ≠ s do
32:         mom.PreviousEdge.AddFlow(flussoInviabile)
33:         if u_f(mom.PreviousEdge) = 0 then
34:             resetSource ← true
35:         end if
36:         mom ← mom.PreviosNode
37:     end while
38:     mom ← n
39:     while mom ≠ t do
40:         mom.NextEdge.AddFlow(flussoInviabile)
41:         if u_f(mom.NextEdge) = 0 then
42:             resetSink ← true
43:         end if
44:         mom ← mom.NextNode
45:     end while
46:     fMax ← fMax + flussoInviabile
47: end while
48: return fMax
```

Algorithm 14 Ricerca di un cammino aumentante Bidirezionale senza alcuna ottimizzazione

Require: rete (G, u, s, t), booleani *sourceSide* e *sinkSide*
Ensure: nodo di confine contenente le informazioni per raggiungere s e t e la quantità di flusso inviabile nel percorso descritto

```
1: codaSource ← coda di nodi
2: codaSink ← coda di nodi
3: if sourceSide then
4:     for all n ∈ V(G)|n.sourceSide do
5:         n.Reset()                          ▷ indico che il nodo non è stato visitato
6:     end for
7:     codaSource.enqueue(s)
8: end if
9: if sinkSide then
10:    for all n ∈ V(G)|¬n.sourceSide do
11:        n.Reset()
12:    end for
13:    codaSink.enqueue(t)
14: end if
15: Procedo con l'esplorazione come descritto nelle appendici 10, 11,12
```

A.6 Pseudo-codice algoritmo bidirezionali di ricerca del flusso massimo con ottimizzazione sugli ultimi livelli e con propagazione di malattia

Algorithm 15 Invio del flusso e inizializzazione delle variabili per Ottimizzazione agli ultimi livelli e propagazione della malattia Bidirezionali

Require: rete (G, u, s, t), pile di nodi $vuotiSource$ e $vuotiSink$
Ensure: quantità di flusso inviata, grafo dei residui aggiornato

1: $vuotiSource \leftarrow$ pila di nodi contenente il nodo s
2: $vuotiSource \leftarrow$ pila di nodi contenente il nodo t
3: $fMax \leftarrow 0$
4: **while** true **do**
5: $n \leftarrow$ ricerca del cammino aumentante $(G, vuotiSource, vuotiSink)$ ▷ si faccia riferimento alle appendici 3.3 e 3.4
6: **if** $n =$ null **then**
7: **break**
8: **end if**
9: $flussoInviabile \leftarrow +\infty$
10: $mom \leftarrow n$
11: **while** $mom \neq s$ **do**
12: **if** $mom.PreviousEdge.Reversed$ **then**
13: $flussoInviabile \leftarrow \min(flussoInviabile, f(mom.PreviosEdge))$
14: **else**
15: $flussoInviabile \leftarrow \min(flussoInviabile, u_f(mom.PreviosEdge))$
16: **end if**
17: $mom \leftarrow mom.PreviosNode$
18: **end while**
19: $mom \leftarrow n$
20: **while** $mom \neq t$ **do**
21: **if** $mom.NextEdge.Reversed$ **then**
22: $flussoInviabile \leftarrow \min(flussoInviabile, f(mom.NextEdge))$
23: **else**
24: $flussoInviabile \leftarrow \min(flussoInviabile, u_f(mom.NextEdge))$
25: **end if**
26: $mom \leftarrow mom.NextEdge$
27: **end while**
28: $vuotiSource.Clear$
29: $vuotiSink.Clear$

```
30:     if flussoInviabile = 0 then
31:         vuotiSource.Push(s)
32:         vuotiSink.Push(t)
33:         for all n ∈ V(G) \ s, t do
34:             n.Visited ← false
35:         end for
36:         continue
37:     end if
38:     mom ← n
39:     while mom ≠ s do
40:         mom.PreviousEdge.AddFlow(flussoInviabile)
41:         if u_f(mom.PreviousEdge) = 0 then
42:             vuotiSource.Push(mom)
43:             mom.SetValid(false)
44:         end if
45:         mom ← mom.PreviousNode
46:     end while
47:     mom ← n
48:     while mom ≠ t do
49:         mom.NextEdge.AddFlow(flussoInviabile)
50:         if u_f(mom.NextEdge) = 0 then
51:             vuotiSource.Push(mom)
52:             mom.SetValid(false)
53:         end if
54:         mom ← mom.NextNode
55:     end while
56:     fMax ← fMax + flussoInviabile
57: end while
58: return fMax
```

Algorithm 16 Ricerca di un cammino aumentante ottimizzazione negli ultimi livelli bidirezionale

Require: rete (G, u, s, t), e due pile *vuotiSource* e *vuotiSink*
Ensure: nodo di confine contenente le informazioni per raggiungere s e t

```
1: if ¬vuotiSource.isEmpty then
2:     repaired ← true
3:     while ¬vuotiSource.isEmpty do
4:         noCapSource ← vuotiSource.pop()
5:         if ¬ Repair(G, noCapSource,false) then          ▷ si faccia riferimento all'algoritmo 20
6:             vuotiSource.Push(noCapSource)
7:             repaired ← false
8:             break
9:         end if
10:    end while
11:    if vuotiSink.isEmpty ∧ repaired then
12:        for all n ∈ LastNodesSinkSide |n.valid  do
13:            if Reached(s, n) then  ▷ da n, indico se riesco a raggiungere il nodo s attraverso PreviousNode o NextNode
14:                return n
15:            end if
16:        end for
17:    end if
18:    if ¬repaired then
19:        if noCapSource = s then
20:            codaSource.enqueue(s)
21:        else if noCapSource ∈ LastSinkNodes then
22:            for all n ∈ LastNodesSourceSide do
23:                codaSource.enqueue(n)
24:            end for
25:        else
26:            for all n ∈ V(G)|n.SourceSide ∧ n.label + 1 = noCapSource.label do
27:                codaSource.enqueue(n)
28:            end for
29:            for all n ∈ V(G)|n.SourceSide ∧ n.label ≥ noCapSource.label do
30:                n.Visited ← false
31:            end for
32:        end if
33:    end if
34: end if
```

```
35: if ¬vuotiSink.isEmpty then
36:     repaired ← true
37:     while ¬vuotiSink.isEmpty do
38:         noCapSink ← vuotiSink.pop()
39:         if ¬Repair(G, noCapSink,true) then
40:             vuotiSink.push(noCapSink)
41:             repaired ← false
42:             break
43:         end if
44:     end while
45:     if repaired ∧ vuotiSource.isEmpty then
46:         for all n ∈ LastNodesSinkSide|n.valid do
47:             if Reached(t, n) then
48:                 return n
49:             end if
50:         end for
51:     end if
52:     if ¬repaired then
53:         if noCapSink = t then
54:             codaSink.enqueue(t)
55:         else
56:             for all n ∈ V(G)|¬n.SourceSide ∧ n.label + 1 = noCapSink.label do
57:                 codaSink.enqueue(n)
58:             end for
59:             for all n ∈ V(G)|¬n.SourceSide ∧ n.label ≥ noCapSink.label do
60:                 n.Visited ← false
61:             end for
62:         end if
63:     end if
64: end if
65: Procedo con l'esplorazione come descritto nelle appendici 10, 11,12
```

Algorithm 17 Ricerca di un cammino aumentante con propagazione della malattia bidirezionale

Require: rete (G, u, s, t), e due pile *vuotiSource* e *vuotiSink*
Ensure: nodo di confine contenente le informazioni per raggiungere s e t

```
1: sourceRepaired ← vuotiSource.isEmpty
2: sinkRepaired ← vuotiSink.isEmpty
3: if ¬vuotiSource.isEmpty then
4:     repaired ← true
5:     momNoCap ← null
6:     while ¬vuotiSource.isEmpty do
7:         noCapSource ← vuotiSource.pop()
8:         if ¬ Repair(G, noCapSource,false) then          ▷ si faccia riferimento all'algoritmo 20
9:             malati.enqueue(noCapSource)
10:            repaired ← false
11:            if momNoCap = null then
12:                momNoCap ← noCapSource
13:            end if
14:        end if
15:    end while
16:    noCapSource ← momNoCap
17:    if sinkRepaired ∧ repaired then
18:        for all n ∈ LastNodesSinkSide |n.valid  do
19:            if Reached(s, n) then  ▷ da n, indico se riesco a raggiungere il nodo s attraverso PreviousNode o NextNode
20:                return n
21:            end if
22:        end for
23:    end if
24:    malato ← null
25:    min ← +∞
26:    while ¬malati.isEmpty do
27:
28:        (momNode, momMin) ← SourceSickPropagation(malati.Dequeue()) ▷ si faccia riferimento all'algoritmo 18
29:        if malato =null then
30:            malato ← momNode
31:        end if
32:        min ← min(min, momMin)
33:    end while
```

34: **if** $malato \neq null \wedge sinkRepaired$ **then**
35: **return** malato
36: **end if**
37: **if** $sinkRepaired$ **then**
38: **for all** $n \in V(G) | n.PreviousNode \neq null \wedge n.NextNode \neq null \wedge n.SourceValid \wedge n.SinkValid \wedge ((n.NextEdge.Reversed \wedge f(n.NextEdge) > 0) \vee (\neg n.NextEdge.Reversed \wedge u_f(n.NextEdge) > 0)) \wedge n.NextNode.Visited \wedge n.NextNode.SinkValid$ **do**
39: **if** Reachable(s, n) **then**
40: **return** n
41: **end if**
42: **end for**
43: **end if**
44: $sourceRepaired \leftarrow repaired$
45: **if** $\neg repaired$ **then**
46: **if** $noCapSource = s$ **then**
47: $codaSource$.enqueue(s)
48: **else if** $noCapSource \in LastSinkNodes$ **then**
49: **for all** $n \in LastNodesSourceSide$ **do**
50: $codaSource$.enqueue(n)
51: **end for**
52: **else if** $min = +\infty$ **then**
53: **for all** $n \in V(G) | n.SourceSide \wedge n.label + 1 = noCapSource.label$ **do**
54: $codaSource$.enqueue(n)
55: **end for**
56: **for all** $n \in V(G) | n.SourceSide \wedge n.label \geq noCapSource.label$ **do**
57: $n.Visited \leftarrow$ false
58: **end for**
59: **else**
60: **for all** $n \in V(G) | n.SourceSide \wedge n.label = min$ **do**
61: $codaSource$.enqueue(n)
62: **end for**
63: **for all** $n \in V(G) | n.SourceSide \wedge n.label \geq min + 1$ **do**
64: $n.Visited \leftarrow$ false
65: **end for**
66: **end if**
67: **end if**
68: **end if**

69: **if** $\neg vuotiSink.isEmpty$ **then**
70: $repaired \leftarrow$ true
71: **while** $\neg vuotiSink.isEmpty$ **do**
72: $noCapSink \leftarrow vuotiSink$.pop()
73: **if** $\neg$Repair($G, noCapSink$,true) **then**
74: $vuotiSink$.push($noCapSink$)
75: $repaired \leftarrow$ false
76: **break**
77: **end if**
78: **end while**
79: **if** $repaired \wedge vuotiSource.isEmpty$ **then**
80: **for all** $n \in LastNodesSinkSide | n.valid$ **do**
81: **if** Reached(t, n) **then**
82: **return** n
83: **end if**
84: **end for**
85: **end if**
86: $malato \leftarrow$ null
87: $min \leftarrow +\infty$
88: **while** $\neg malati.isEmpty$ **do**
89: $(momNode, momMin) \leftarrow$ SinkSickPropagation($malati$.Dequeue()) ▷ si faccia riferimento all'algoritmo 19
90: **if** $malato =$null **then**
91: $malato \leftarrow momNode$
92: **end if**
93: $min \leftarrow \min(min, momMin)$
94: **end while**
95: **if** $malato \neq null$ **then**
96: **return** malato
97: **end if**
98: **if** $sinkRepaired$ **then**
99: **for all** $n \in V(G) | n.PreviousNode \neq null \wedge n.NextNode \neq null \wedge n.SourceValid \wedge n.SinkValid \wedge ((n.PreviousEdge.Reversed \wedge f(n.PreviousEdge) > 0) \vee (\neg n.PreviousEdge.Reversed \wedge u_f(n.PreviousEdge) > 0)) \wedge n.PreviousNode.Visited$ **do**
100: **if** Reachable(t, n) **then**
101: **return** n
102: **end if**
103: **end for**
104: **end if**

```
105:     sinkRepaired ← repaired
106:     if ¬repaired then
107:         if noCapSink = t then
108:             codaSink.enqueue(t)
109:         else if min = +∞ then
110:             for all n ∈ V(G)|¬n.SourceSide ∧ n.label + 1 = noCapSink.label do
111:                 codaSink.enqueue(n)
112:             end for
113:             for all n ∈ V(G)|¬n.SourceSide ∧ n.label ≥ noCapSink.label do
114:                 n.Visited ← false
115:             end for
116:         else
117:             for all n ∈ V(G)|¬n.SourceSide ∧ n.label = min do
118:                 codaSink.enqueue(n)
119:             end for
120:             for all n ∈ V(G)|¬n.SourceSide ∧ n.label ≥ min do
121:                 n.Visited ← false
122:             end for
123:         end if
124:     end if
125: end if
126: Procedo con l'esplorazione 10, 11 o 12
```

Algorithm 18 Propagazione della malattia dalla parte di Source

Require: nodo non riparabile *node*
Ensure: una coppia (nodo,intero)

1: $min \leftarrow +\infty$
2: $malati \leftarrow$ coda di nodi vuota
3: $malati$.enqueue($node$)
4: **while** $\neg malati.isEmpty$ **do**
5: $m \leftarrow malati$.dequeue()
6: **if** $m.SourceSide \vee m \in LastNodesSinkSide$ **then**
7: **if** $\neg RepairNode(m, false)$ **then** ▷ vedesi algoritmo 20
8: **for all** $edge \in \delta^+(m) | edge = edge.NextNode.PreviousEdge$ **do**
9: $malati.enqueue(e.NextNode)$
10: **end for**
11: **for all** $edge \in \delta^-(m) | edge = edge.PreviousEdge.PreviousEdge$ **do**
12: $malati.enqueue(e.NextNode)$
13: **end for**
14: **else if** $m.NextEdge \neq null \wedge ((m.NextEdge.Reversed \wedge f(m.NextEdge) > 0) \vee (\neg m.NextEdge.Reversed \wedge u_f(m.NextEdge) > 0)) \wedge m.NextNode.Visited \wedge m.SinkValid$ **then**
15: **return** (m,min)
16: **else if** $m.SourceSide$ **then**
17: $min = \min(min, m.Label)$
18: **end if**
19: **end if**
20: **end while**
21: $(null, min)$

Algorithm 19 Propagazione della malattia dalla parte di Sink

Require: nodo non riparabile *node*
Ensure: una coppia (nodo,intero)

1: $min \leftarrow +\infty$
2: $malati \leftarrow$ coda di nodi vuota
3: $malati$.enqueue($node$)
4: **while** $\neg malati.isEmpty$ **do**
5: $m \leftarrow malati$.dequeue()
6: **if** $\neg m.SourceSide$ **then**
7: **if** $\neg RepairNode(m, true)$ **then** ▷ vedesi algoritmo 20
8: **for all** $edge \in \delta^+(m) | edge = edge.NextNode.NextEdge$ **do**
9: $malati.enqueue(e.NextNode)$
10: **end for**
11: **for all** $edge \in \delta^-(m) | edge = edge.PreviousEdge.NextEdge$ **do**
12: $malati.enqueue(e.NextNode)$
13: **end for**
14: **else if** $m.PreviousEdge \neq null \wedge ((m.PreviousEdge.Reversed \wedge f(m.PreviousEdge) > 0) \vee (\neg m.PreviousEdge.Reversed \wedge u_f(m.PreviousEdge) > 0)) \wedge m.PreviousNode.Visited \wedge m.SourceValid$ **then**
15: **return** (m,min)
16: **else**
17: $min = \min(min, m.Label)$
18: **end if**
19: **end if**
20: **end while**
21: $(null, min)$

Algorithm 20 Riparazione di un nodo per algoritmi bidirezionali

Nel caso si usi la riparazione del nodo con ottimizzazione negli ultimi livelli, SourceValid e SinkValid valgono come Valid

Require: rete (G, u, s, t), nodo da riparare *node*, booleano *onlySinkExploration*, che mi indica, nel caso di un nodo di confine, se devo esplorare solo i nodi esplorati da source o solo quelli esplorati da sink

Ensure: booleano che indica se il nodo è stato riparato o meno

1: **if** $node = s \vee node = t$ **then**
2: **return** false
3: **end if**
4: **if** $node.PreviousEdge \neq null \wedge node.NextEdge \neq null \wedge node.PreviousNode.SourceValid \wedge node.NextNode.SinkValid \wedge ((node.PreviousEdge.Reversed \wedge f(node.PreviousEdge) > 0) \vee (\neg node.PreviousEdge.Reversed \wedge u_f(node.PreviousEdge) > 0)) \wedge ((node.NextEdge.Reversed \wedge f(node.NextEdge) > 0) \vee (\neg node.nextEdge.Reveresed \wedge u_f(node.NextEdge) > 0))$ **then**
5: **return** true ▷ il nodo è di confine ed è già riparato
6: **else if** $node.PreviousEdge \neq null \wedge node.PreviousNode.SourceValid \wedge ((node.PreviousEdge.Reversed \wedge f(node.PreviousEdge) > 0) \vee (\neg node.PreviousEdge.Reversed \wedge u_f(node.PreviousEdge) > 0) \wedge node.NextEdge = null)$ **then**
7: **return** true ▷ il nodo è stato esplorato da source ed è già riparato
8: **else if** $node.NextEdge \neq null \wedge node.NextNode.SinkValid \wedge ((node.NextEdge.Reversed \wedge f(node.NextEdge) > 0) \vee \neg node.NextEdge.Reversed \wedge u_f(node.NextEdge) > 0) \wedge node.PreviousEdge = null)$ **then**
9: **return** true ▷ il nodo è stato esplorato da sink ed è già riparato
10: **end if**
11: **if** *node.SourceSide* **then**
12: **for all** $edge \in \delta^+(node)$ **do**
13: $n \leftarrow edge.NextNode$
14: **if** $node.SourceSide = n.SourceSide \wedge f(edge) > 0 \wedge n.Label = node.label - 1 \wedge n.SourceValid \wedge n.Visited \wedge n.PreviousNode \neq node$ **then**
15: *node*.update$(n, edge)$
16: **return** true
17: **end if**
18: **end for**
19: **for all** $edge \in \delta^-(node)$ **do**
20: $p \leftarrow edge.PreviousNode$

21: **if** $node.SourceSide = p.SourceSide \wedge u_f(edge) > 0 \wedge p.Label = node.label - 1 \wedge p.SourceValid \wedge p.Visited \wedge p.PreviousNode \neq node$ **then**
22: $node$.update$(p, edge)$
23: **return** true
24: **end if**
25: **end for**
26: **else**
27: **for all** $edge \in \delta^+(node)$ **do**
28: $n \leftarrow edge.NextNode$
29: **if** $node.SourceSide \neq n.SourceSide \wedge \neg onlySinkExploration \wedge f(edge) > 0 \wedge n.SourceValid \wedge n.SourceSide \wedge n.Visited$ **then**
30: $node$.update$(n, edge)$
31: **return** true
32: **else if** $n.SourceSide = node.SourceSide \wedge onlySinkExploration \wedge u_f(edge) > 0 \wedge n.SinkValid \wedge node.Label = n.Label + 1 \wedge n.Visited \wedge n.NextNode \neq node$ **then**
33: $node$.update$(n, edge)$
34: **return** true
35: **end if**
36: **end for**
37: **for all** $edge \in \delta^-(node)$ **do**
38: $p \leftarrow edge.PreviousNode$
39: **if** $node.SourceSide \neq p.SourceSide \wedge \neg onlySinkExploration \wedge u_f(edge) > 0 \wedge p.SourceValid \wedge p.SourceSide \wedge p.Visited$ **then**
40: $node$.update$(p, edge)$
41: **return** true
42: **else if** $p.SourceSide = node.SourceSide \wedge onlySinkExploration \wedge f(edge) > 0 \wedge p.SinkValid \wedge node.Label = p.Label + 1 \wedge p.Visited \wedge p.NextNode \neq node$ **then**
43: $node$.update$(p, edge)$
44: **return** true
45: **end if**
46: **end for**
47: **end if**
48: **if** $node.SourceSide \vee \neg onlySinkExploration$ **then**
49: $node.SourceValid \leftarrow$ false
50: **else**
51: $node.SinkValid \leftarrow$ false
52: **end if**
53: **return** false

A.7 Shortest Augmenting Path bidirezionale

Algorithm 21 Inizializzazione ed Augment

Require: rete (G, u, s, t)
Ensure: quantità di flusso inviata, grafo dei residui aggiornato

1: $fMax \leftarrow$ Bfs(s) ▷ faccio partire da s una bfs, cercando un percorso e sopratutto indicando la distanza d_s
2: sendFlow$(t, fMax)$ ▷ invio il flusso dal percorso indicato tramite previousNode da t verso s con il valore fMax,nel mentre che procedo cancello le informazioni nei nodi esplorati (tranne la distanza)
3: $f \leftarrow$ Bfs(t) ▷ bfs da t verso s, trovo un percorso salvato da NextNode e sopratutto trovo la distanza d_t
4: sendFlow(s, f)
5: $fMax \leftarrow f + fMax$
6: **for all** $n \in V(G)$ **do**
7: n.Reset() ▷ cancello indicazioni su un possibile percorso da fare
8: **end for**
9: $fso \leftarrow +\infty, fsi \leftarrow +\infty$
10: **while** $f \neq 0 \wedge d_s(t) > \#V(G) \wedge d_t(s) > \#V(G)$ **do**
11: $(fso, fsi, startSource, startSink) \leftarrow$ SourceDfs (G, $startSource$, $startSink$, fso, fsi, $codaSource$, $codaSink$) ▷ vedesi algoritmo 22
12: **if** $startSink = startSource \wedge startSink \neq null$ **then**
13: $f \leftarrow \min(fso, fsi)$
14: sendFlow$(startSink, f)$
15: $fso \leftarrow +\infty, fsi \leftarrow +\infty$
16: $startSource \leftarrow s, startSink \leftarrow t$
17: **while** $\neg codaSource.isEmpty$ **do**
18: $codaSource$.dequeue().Reset()
19: **end while**
20: **while** $\neg codaSink.isEmpty$ **do**
21: $codaSink$.dequeue().Reset()
22: **end while**
23: **else if** $startSink = s$ **then**
24: $f \leftarrow fsi$
25: sendFlow$(startSink, f)$
26: $fsi \leftarrow +\infty$
27: $startSink \leftarrow t$
28: **while** $\neg codaSink.isEmpty$ **do**
29: $codaSink$.dequeue().Reset()
30: **end while**

```
31:     else if startSource = t then
32:         f ← fso
33:         sendFlow(startSource, f)
34:         fso ← +∞
35:         startsource ← s
36:         while ¬codaSource.isEmpty do
37:             codaSource.dequeue().Reset()
38:         end while
39:     else
40:         break
41:     end if
42:     fMax ← f + fMax
43: end while
44: return fMax
```

Algorithm 22 Ricerca di un cammino aumentante della parte di Source

Require: rete (G, u, s, t), nodi di partenza *startSource* e *startSink*, valore del flusso per parte *sourceFlow* e *sinkFlow*,code di nodi esplorati *codaSource* e *codaSink*

Ensure: quantità di flusso inviabile per source e per sink, ultimo nodo esplorato da parte di source e di sink

1: **if** $startSource = startSink$ **then**
2: **return** $(sourceFlow, sinkFlow, startSource, startSink)$
3: **end if**
4: **if** $d_s(startSink) < \#V(G) \wedge d_t(startSource) < \#V(G)$ **then**
5: **for all** $edge \in \delta^+(startSource) | u_f(edge) > 0 \wedge d_t(edge.NextNode) = d_t(startSource) - 1$ **do**
6: $n \leftarrow edge.NextNode$
7: $sourceFlow \leftarrow \min(sourceFlow, u_f(edge))$
8: $n.previousEdge \leftarrow edge$
9: $n.PreviousNode \leftarrow sourceFlow$
10: $codaSource$.enqueue(n)
11: **if** $n = t$ **then**
12: **return** $(sourceFlow, sinkFlow, n, startSink)$
13: **end if**
14: **if** $n.nextEdge \neq null$ **then**
15: **return** $(sourceFlow, sinkFlow, n, n)$
16: **end if**
17: **return** SinkDfs(G, n, $startSink$, $sourceFlow$, $sinkFlow$, $codaSource$, $codaSink$) ▷ vedesi algoritmo 23
18: **end for**
19: $minDistance \leftarrow +\infty$
20: **for all** $edge \in \delta^+(startSource) | u_f(edge)$ **do**
21: $minDistance \leftarrow \min(minDistance, d_t(edge.nextNode))$
22: **end for**
23: $d_t(startSource) \leftarrow minDistance + 1$
24: **if** $startSource = s$ **then**
25: $mom \leftarrow startsource$
26: **else**
27: $mom \leftarrow startSource.previousNode$
28: **end if**
29: **return** SourceDfs(G, mom, $startSink$, $sourceFlow, sinkFlow$, $codaSource$, $codaSink$) ▷ vedasi algoritmo 22
30: **end if**
31: **return** $(0, 0, null, null)$

Algorithm 23 Ricerca di un cammino aumentante della parte di Sink

Require: rete (G, u, s, t), nodi di partenza *startSource* e *startSink*, valore del flusso per parte *sourceFlow* e *sinkFlow*,code di nodi esplorati *codaSource* e *codaSink*

Ensure: quantità di flusso inviabile per source e per sink, ultimo nodo esplorato da parte di source e di sink

1: **if** $startSource = startSink$ **then**
2: **return** $(sourceFlow, sinkFlow, startSource, startSink)$
3: **end if**
4: **if** $d_s(startSink) < \#V(G) \wedge d_t(startSource) < \#V(G)$ **then**
5: **for all** $edge \in \delta^-(startSink) | u_f(edge) > 0 \wedge d_s(edge.PreviousNode) = d_s(startSink) - 1$ **do**
6: $p \leftarrow edge.PreviousNode$
7: $sourceFlow \leftarrow \min(sinkFlow, u_f(edge))$
8: $codaSink$.enqueue$(edge.PreviousNode)$
9: $p.NextEdge \leftarrow edge$
10: $p.NextNode \leftarrow startSink$
11: **if** $p = s$ **then**
12: **return** $(sourceFlow, sinkFlow, startSource, p)$
13: **end if**
14: **if** $p.previousNode \neq null$ **then**
15: **return** $(sourceFlow, sinkFlow, p, p)$
16: **end if**
17: **return** SourceDfs$G$, $startSource$, p, $sourceFlow, sinkFlow$, s, t, $codaSource$, $codaSink$) ▷ vedasi algoritmo 22
18: **end for**
19: $minDistance \leftarrow +\infty$
20: **for all** $edge \in \delta^-(starSink) | u_f(edge) > 0$ **do**
21: $minDistance \leftarrow \min(minDistance, d_s(edge.previousNode)$
22: **end for**
23: $d_s(startSink) \leftarrow minDistance + 1$
24: **if** $startSink = t$ **then**
25: $mom \leftarrow startSink$
26: **else**
27: $mom \leftarrow startSink.nextNode$
28: **end if**
29: **return** SinkDfs(G, $startSource$, mom, $sourceFlow$, $sinkFlow$, $codaSource$, $codaSink$) ▷ vedasi algoritmo 23
30: **end if**
31: **return** $(0, 0, null, null)$

A.8 Pseudo-codice creazione del grafo

Algorithm 24 creazione del grafo

Require: valore della cardinalità del grafo -1, chiamata *cardinalitá*
Ensure: rete (G, u, s, t)

1: $lista \leftarrow$ lista di nodi vuota
2: $grafo \leftarrow$ oggetto grafo
3: $s \leftarrow$ nodo sorgente, con nome 0
4: $lista$.Enqueue(s)
5: $grafo$.AddNode(s)
6: $i \leftarrow 1$
7: **while** $i <$ *cardinalitá* **do**
8: $n \leftarrow$ nodo, con nome i
9: $lista$.Enqueue(n)
10: $grafo$.AddNode(n)
11: $i \leftarrow i + 1$
12: **end while**
13: $t \leftarrow$ nodo destinazione, con nome pari a *cardinalitá*
14: $lista$.Enqueue(t)
15: $grafo$.AddNode(t)
16: $i \leftarrow 0$
17: **while** $i <$ *cardinalitá* **do**
18: $node \leftarrow lista[i]$
19: $numArc \leftarrow Rand(1, (cardinalitá - i)\%(cardinalitá/10))$ ▷ con Rand si intende una funzione random, con primo elemento limite inferiore e secondo elemento limite superiore, entrambi compressi
20: $x \leftarrow i + 1$
21: **while** $x \leq i + numArc$ **do**
22: $dest \leftarrow lista[x]$
23: $cap \leftarrow Rand(0, 9999)$ ▷ la scelta del limite superiore della capacità è arbitraria, cap indica la capacità dell'arco che si sta creando
24: **if** $cap > 0$ **then**
25: $edge \leftarrow$ arco $(node, dest)$ con capacità cap,
26: $node$.AddEdge($edge$)
27: $dest$.AddEdge($edge$)
28: **end if**
29: $x \leftarrow x + 1$
30: **end while**
31: $i \leftarrow i + 1$
32: **end while**
33: **return** grafo

Appendice B

Tabelle

Per questioni di spazio, si è preferito accorciare i nomi degli algoritmi come segue:

- Seed è il valore usato per inizializzare la funzione random durante la creazione del grafo
- Flow indica il flusso inviato tramite quel grafo
- Nodi è il numero di nodi che ha il grafo
- Archi è il numero di archi che ha il grafo
- NoOpt si intende il tempo impiegato,in ms, per la risoluzione con l'algoritmo senza nessuna ottimizzazione, l'algoritmo monodirezionale è stato presentato nel capitolo 2.1, l'algoritmo bidirezionale nel capitolo 3.2
- LLO si intende il tempo impiegato, in ms, per la risoluzione con l'algoritmo con ottimizzazione sugli ultimi livelli, l'algoritmo monodirezionale è stato presentato nel capitolo 2.2, l'algoritmo bidirezionale nel capitolo 3.3
- SP si intende il tempo impiegato,in ms, per la risoluzione con l'algoritmo con propagazione della malattia, l'algoritmo monodirezionale è stato presentato nel capitolo 2.3, l'algoritmo bidirezionale nel capitolo 3.4
- SAP si intende il tempo,in ms, impiegato per la risoluzione con l'algoritmo Shortest Augmenting Path,l'algoritmo monodirezionale è stato presentato nel capitolo 2.4, l'algoritmo bidirezionale nel capitolo 3.5.

Seed	Flow	Nodi	Archi	NoOpt	LLO	SP	SAP
704671965	74658	10001	4538924	14436	2731	262	1076558
1779933806	39083	10001	4621050	6843	604	164	1130075
1837959025	25789	10001	4589324	4927	904	162	1107433
530927106	49384	10001	4579878	7990	1022	149	1104947
1108985004	36241	10001	4546812	7200	326	175	1085228
56633772	12157	10001	4580108	2322	163	154	1104465
1664018830	26777	10001	4589879	5719	1021	154	1103199
380361422	30601	10001	4522941	5392	994	152	1077939
923436165	31665	10001	4558172	6603	1184	145	1099867
483222406	32903	10001	4528804	6686	1783	263	1096112
365901078	24775	10001	4533518	4408	166	152	1086252
1438274553	53369	10001	4518001	9048	1098	142	1082656
1360147033	41449	10001	4507137	7353	1002	191	1068807
551995700	49974	10001	4573678	9726	2020	274	1082977
1077814831	40248	10001	4528814	8649	1439	175	1082421
463179990	36858	10001	4542122	7667	1413	300	1063181
1598168404	27929	10001	4561478	5451	706	142	1088484
708192553	34501	10001	4556167	6958	587	293	1089199
1587934532	51289	10001	4559401	10158	1964	155	1116352
2011329856	46736	10001	4537840	8588	1576	155	1094953

Tabella 2: tempi degli algoritmi monodirezionali

Seed	Flow	Nodi	Archi	NoOpt	LLO	SP
704671965	74658	10001	4538924	6507	2448	897
1779933806	39083	10001	4621050	3385	1294	760
530927106	49384	10001	4579878	3582	1763	965
1837959025	25789	10001	4589324	2817	1871	556
1108985004	36241	10001	4546812	3685	829	666
56633772	12157	10001	4580108	1128	367	390
1664018830	26777	10001	4589879	2501	1038	758
380361422	30601	10001	4522941	2528	1103	664
923436165	31665	10001	4558172	2664	1253	901
483222406	32903	10001	4528804	3007	1402	782
365901078	24775	10001	4533518	1848	648	706
1360147033	41449	10001	4507137	3251	1196	746
1438274553	53369	10001	4518001	3870	1254	779
551995700	49974	10001	4573678	4536	1518	707
463179990	36858	10001	4542122	3648	1335	680
1077814831	40248	10001	4528814	4154	1516	945
1598168404	27929	10001	4561478	2266	945	567
708192553	34501	10001	4556167	2843	1020	568
1587934532	51289	10001	4559401	5523	1515	639
2011329856	46736	10001	4537840	4328	1792	692

Tabella 3: Tempo degli algoritmi bidirezionali con propagazione dei nodi

Seed	Flow	Nodi	Archi	NoOpt	LLO	SP
704671965	74658	10001	4538924	11766	4331	1184
1779933806	39083	10001	4621050	5794	1972	911
1837959025	25789	10001	4589324	3934	1783	750
530927106	49384	10001	4579878	6361	3129	1393
1108985004	36241	10001	4546812	5741	2036	864
56633772	12157	10001	4580108	2055	674	544
1664018830	26777	10001	4589879	4231	1838	982
380361422	30601	10001	4522941	4607	1936	867
923436165	31665	10001	4558172	4967	2112	1196
483222406	32903	10001	4528804	5576	2656	1000
365901078	24775	10001	4533518	3455	1389	971
1438274553	53369	10001	4518001	6845	2356	1054
1360147033	41449	10001	4507137	5981	2081	1024
551995700	49974	10001	4573678	8283	2670	899
1077814831	40248	10001	4528814	7469	3130	1266
463179990	36858	10001	4542122	6362	2326	885
1598168404	27929	10001	4561478	4193	1516	614
708192553	34501	10001	4556167	5151	1717	777
1587934532	51289	10001	4559401	9010	2620	918
2011329856	46736	10001	4537840	7452	2884	907

Tabella 4: Tempo degli algoritmi bidirezionali esplorando un nodo per ogni parte alla volta

Seed	Flow	Nodi	Archi	NoOpt	LLO	SP
704671965	74658	10001	4538924	7093	2333	762
1779933806	39083	10001	4621050	3379	853	717
1837959025	25789	10001	4589324	2691	1394	529
530927106	49384	10001	4579878	3845	1408	740
1108985004	36241	10001	4546812	3658	763	746
56633772	12157	10001	4580108	1201	289	327
1664018830	26777	10001	4589879	2553	985	682
3803614	30601	10001	4522941	1554	863	593
923436165	31665	10001	4558172	2755	1213	600
483222406	32903	10001	4528804	3227	1443	625
365901078	24775	10001	4533518	2077	486	529
1438274553	53369	10001	4518001	4401	1067	547
1360147033	41449	10001	4507137	3434	986	620
551995700	49974	10001	4573678	5095	1466	651
1077814831	40248	10001	4528814	4353	1329	681
463179990	36858	10001	4542122	3907	1367	635
1598168404	27929	10001	4561478	2347	884	623
708192553	34501	10001	4556167	3270	770	631
1587934532	51289	10001	4559401	5673	1450	714
2011329856	46736	10001	4537840	4346	1304	541

Tabella 5: Tempo degli algoritmi bidirezionali con esplorazione per stessa label

Seed	Flow	Nodi	Archi	Tempo
704671965	74658	10001	4538924	180
1779933806	39083	10001	4621050	187
1837959025	25789	10001	4589324	172
530927106	49384	10001	4579878	191
1108985004	36241	10001	4546812	177
56633772	12157	10001	4580108	169
1664018830	26777	10001	4589879	184
380361422	30601	10001	4522941	184
923436165	31665	10001	4558172	177
483222406	32903	10001	4528804	184
365901078	24775	10001	4533518	165
1438274553	53369	10001	4518001	183
1360147033	41449	10001	4507137	179
551995700	49974	10001	4573678	193
1077814831	40248	10001	4528814	187
463179990	36858	10001	4542122	216
1598168404	27929	10001	4561478	185
708192553	34501	10001	4556167	173
1587934532	51289	10001	4559401	194
2011329856	46736	10001	4537840	190

Tabella 6: Tempo per l'algoritmo Shortest Augmenting Path bidirezionale