Il lavoro svolto in questa tesi è stato quello di studiare ed analizzare i vari algoritmi che dai primi anni sessanta, e in modo sempre più efficiente, fino alla soglie del duemila sono stati proposti per la risoluzione di uno dei problemi di ottimizzazione tra i più studiati in informatica teorica: il problema del flusso massimo. Questo problema consiste nel determinare qual è il massimo flusso che può transitare in una data rete, ossia, in altri termini, quale è la massima quantità di informazione, di fluido o di altro materiale che può transitare in una rete di trasporto o di comunicazione nell'unità di tempo.

Per rete di flusso intenderemo un grafo $G = (V, E)$ orientato, in cui ad ogni arco è associato un valore intero detto capacità (indicheremo con $c(u, v)$ la capacità dell'arco (u, v)); l'obiettivo è quello di inviare più flusso possibile tra due vertici speciali, la sorgente s ed il pozzo t, senza violare i vincoli di capacità degli archi.

Un **flusso** in G è una funzione a valori reali $f : V \times V \to \mathbf{R}$ che soddisfa le seguenti tre proprietà:

1. **Vincolo di capacità**: Per tutti gli $u, v \in V$, si richiede $f(u, v) \leqslant c(u, v)$.
2. **Antisimmetria**: Per tutti gli $u, v \in V$, si richiede $f(u, v) = -f(v, u)$.
3. **Conservazione del flusso**: $\forall u \in V - \{s, t\}$

$$\sum_{v \in V} f(u, v) = 0.$$

Diremo l'arco (u, v) **saturo** se $f(u, v) = c(u, v)$, cioè se inviando altro flusso attraverso l'arco (u, v) si violano i vincoli di capacità. Inoltre definiamo il **valore** di un flusso come $|f| = \sum_{v \in V} f(u, v)$.

La correttezza degli algoritmi studiati si basa sul teorema fondamentale "Flusso Massimo Taglio Minimo". Per analizzare questo teorema abbiamo

introdotto tre importanti elementi: il taglio di una rete di flusso, la rete residua ed il cammino aumentante.

Il **taglio** (S, T) di una rete di flusso $G = (V, E)$ è una partizione di V tale che $s \in S$ e $t \in T$; la **rete residua** è una rete costituita da archi attraverso cui si può inviare flusso senza violare i vincoli di capacità; infine un **cammino aumentate** p è un cammino dalla sorgente al pozzo nella rete residua.

Il teorema "Flusso Massimo Taglio Minimo" collega in modo biunivoco la ricerca del flusso massimo alla determinazione di un taglio di capacità minima nella rete di flusso.

Teorema. *(Flusso Massimo Taglio Minimo)*
Se f è un flusso in una rete di flusso $G = (V, E)$ con sorgente s e pozzo t allora le seguenti condizioni sono equivalenti:

1. *f è un flusso massimo in G.*
2. *La rete residua G_f non contiene cammini aumentanti.*
3. *$|f| = c(S, T)$ per un qualche taglio (S, T) di G.*

Il teorema "Flusso Massimo Taglio Minimo" implica che finché nella rete residua esistono cammini aumentanti si può trovare un flusso di valore maggiore e, una volta che nella rete residua non vi sono più cammini aumentanti il valore del flusso è massimo. Questo teorema, pur essendo molto importante dal punto di vista teorico, non ha però una grande utilità in pratica. Trovare tutti i tagli in una rete di flusso, per determinare quale tra questi sia quello di capacità minima, ha infatti un costo che cresce esponenzialmente al crescere del numero di archi nella rete e quindi un costo eccessivo e poco significativo.

I primi a trovare un metodo che permette di dare una soluzione al problema del flusso massimo in tempo polinomiale furono Ford e Fulkerson (1956)[8].

L'algoritmo iterativo di Ford e Fulkerson opera nel modo seguente. Si parte con $f(u, v) = 0$ per tutti gli spigoli $(u, v) \in E$ ottenendo un flusso iniziale di valore nullo. Ad ogni iterazione si incrementa il valore del flusso cercando un cammino p da s a t nella rete residua e aumentando il flusso lungo questo cammino della sua capacità residua $c_f(p)$. Questo processo viene ripetuto finché non diventa impossibile trovare un cammino aumentante.

Il tempo di esecuzione dell'algoritmo di Ford e Fulkerson dipende da come viene determinato il cammino aumentante p. Se questa scelta non è appropriata l'algoritmo può anche non terminare: il valore del flusso aumenta con incrementi successivi, ma non raggiunge necessariamente un valore di flusso massimo. Un'euristica suggerita da Edmonds e Karp [7][6] per migliorare questa situazione è quella di scegliere sempre un cammino di lunghezza minima.

Nei primi anni novanta Goldberg [12][6] ha proposto un nuovo approccio per il problema del flusso massimo, meglio conosciuto come *algoritmo di preflusso* che, su di una rete di flusso $G = (V, E)$, ha un tempo di esecuzione dell'ordine di $O(|V|^3)$. Gli algoritmi di questo tipo sono i più veloci per risolvere il problema del flusso massimo. Invece di esaminare l'intera rete residua per trovare un cammino aumentante gli algoritmi di preflusso lavorano su di un vertice alla volta. Inoltre non mantengono la proprietà di conservazione del flusso durante la loro esecuzione: tuttavia essi mantengono un **preflusso**, che è una funzione $f : V \times V \to \mathbf{R}$ che soddisfa l'antisimmetria, i vincoli di capacità e il seguente rilassamento della legge di conservazione del flusso:

$$\sum_{v \in V} f(v, u) \geqslant 0$$

per tutti i vertici $u \in V - \{s\}$.

Il flusso netto in un vertice u è chiamato il **flusso in eccesso** in u, ed è dato

da

$$e(u) = \sum_{v \in V} f(v, u).$$

Si dice che un vertice $u \in V - \{s, t\}$ è **traboccante**, oppure analogamente che u è un nodo **attivo**, se $e(u) > 0$.

L'intuizione su cui si basa il metodo di preflusso può essere compresa meglio in termini di flussi liquidi: gli archi corrispondono a tubi e i vertici, che sono i punti di giunzione dei tubi, hanno due interessanti proprietà: la prima è che per accomodare il flusso in eccesso, ogni vertice ha un tubo di scolo che finisce in un serbatoio dove il fluido può essere accumulato; la seconda è che ogni vertice, il suo serbatoio e tutti i tubi ad esso collegati giacciono su di una piattaforma la cui altezza cresce con il progredire dell'algoritmo.

L'altezza di un vertice determina come il flusso viene inviato: infatti il flusso viene inviato solo verso il basso cioè da un vertice più alto ad uno ad altezza inferiore. L'altezza della sorgente è fissata a $|V|$ mentre quella del pozzo è fissata a zero; l'altezza di tutti gli altri vertici parte da zero e aumenta col tempo. L'algoritmo invia quanto più flusso possibile verso il basso, dalla sorgente verso il pozzo. Può succedere che i soli tubi che escono da un vertice u e che non siano ancora saturi di flusso colleghino u a vertici che sono sullo stesso livello di u o più in alto. In questo caso per liberare un vertice traboccante u dal suo flusso in eccesso occorre aumentarne l'altezza tramite un'operazione chiamata *innalzamento* del vertice u. La sua altezza viene aumentata sino ad una unità in più rispetto all'altezza del più basso dei vertici adiacenti verso il quale ha un tubo non saturo. Quindi dopo che il vertice è stato innalzato vi è almeno un tubo uscente da esso attraverso il quale può essere inviato del flusso addizionale.

Per trasformare un preflusso in flusso, l'algoritmo rimanda verso la sorgente il flusso in eccesso raccolto nei serbatoi dei vertici traboccanti: questo viene ottenuto continuando ad innalzare i vertici oltre $|V|$, cioè oltre l'altezza pre-

fissata della sorgente. Una volta che tutti i serbatoi sono stati svuotati, il preflusso non solo è un flusso "legale", ma è anche un flusso massimo.

Di seguito forniamo una codifica delle due operazioni fondamentali: l'invio di eccesso di flusso (che chiameremo PUSH) e l'innalzamento di un vertice (che chiameremo LIFT). L'operazione di base PUSH(u, v) può essere applicata se u è un vertice traboccante, l'arco (u, v) appartiene alla rete residua, e l'altezza del vertice u supera di 1 quella del vertice v.

PUSH(u, v)
1. $d_f(u, v) = \min\{e(u), c(u, v) - f(u, v)\}$
2. $f(u, v) = f(u, v) + d_f(u, v)$
3. $f(v, u) = -f(u, v)$
4. $e(u) = e(u) - d_f(u, v)$
5. $e(v) = e(v) + d_f(u, v)$

LIFT(u)
1. $h(u) = \min\{h(v) \mid (u, v) \in E_f\} + 1$

Il codice per PUSH(u, v) opera nel seguente modo. Si assume che il vertice u abbia un eccesso positivo $e(u)$ e che la capacità residua ($c_f(u, v) = c(u, v) - f(u, v)$) di (u, v) sia positiva. Quindi si può inviare fino a $d_f(u, v) = \min\{e(u), c_f(u, v)\}$ unità di flusso da u a v, senza rischiare di rendere $e(u)$ negativo o di superare la capacità $c(u, v)$. Il flusso viene mosso da u a v aggiornando f ed e. Un'operazione di invio applicata ad un arco (u, v) uscente da un vertice u, si dirà un **invio saturante** se l'arco (u, v) diviene saturo (cioè se dopo l'operazione $c_f(u, v) = 0$ e quindi non compare nella rete residua); altrimenti è un **invio non saturante**.

L'operazione di base LIFT(u) si applica se u è traboccante e se per ogni vertice v per il quale esiste una capacità residua da u a v, non si può inviare del flusso da u a v perchè v non è più basso di u e consiste nell'aumentare l'altezza

di u ($h(u)$) sino ad una unità in più rispetto all'altezza del più basso dei vertici adiacenti verso il quale ha un tubo non saturo. L'algoritmo generico di preflusso utilizza una procedura di inizializzazione che crea un preflusso iniziale f definito da

$$f(u,v) = \begin{cases} c(u,v) & \text{se } u = s, \\ -c(u,v) & \text{se } v = s, \\ 0 & \text{altrimenti.} \end{cases}$$

Quindi ogni arco uscente dalla sorgente viene riempito fino alla capacità mentre tutti gli altri archi non trasportano alcun flusso. Per ogni vertice v adiacente alla sorgente si ha all'inizio $e(v) = c(s,v)$. L'algoritmo generico comincia anche con una funzione altezza h data da

$$h(u) = \begin{cases} |V| & \text{se } u = s, \\ 0 & \text{altrimenti.} \end{cases}$$

L'algoritmo seguente è un tipico esempio del metodo dei preflussi.

INVIO-GENERICO-DI-PREFLUSSO

1. INIZIALIZZAZIONE-PREFLUSSO
2. finché esiste un'operazione di invio o di innalzamento applicabile
seleziona un'operazione di invio o di innalzamento applicabile ed eseguila.

Per dimostrare che l'algoritmo generico di preflusso risolve il problema del flusso massimo, si dimostra prima che, se esso termina, il preflusso f è un flusso massimo; quindi per mostrare che l'algoritmo generico di preflusso effettivamente termina dobbiamo porre un limite superiore al numero delle operazioni che esso può eseguire. Per ognuno dei tre tipi di operazioni abbiamo un limite indipendente: il numero di operazioni di innalzamento è dell'ordine di $O(|V|^2)$, il numero di invii saturanti è al massimo $2|V||E|$ e quello di invii non saturanti è al più $4|V|^2(|V| + |E|)$ per un totale di operazioni di base dell'ordine di $O(|V|^2|E|)$.

È quindi chiaro che il collo di bottiglia dell'algoritmo generico di preflusso è il numero di operazioni di invio non saturante. In realtà esistono vari metodi per esaminare un nodo attivo che possono ridurre il numero di invii non saturanti. Tra questi ne abbiamo esaminati alcuni che riportiamo di seguito.

1. LIFT-TO-FRONT. Esamina i nodi attivi in ordine topologico rispetto ad una certa proprietà ed ha un tempo di esecuzione dell'ordine di $O(|V|^3)$. L'algoritmo, anche detto WAVE è dovuto a Tarjan [2][6] e mantiene una lista contenente tutti i vertici della rete e la scandisce cominciando dalla testa, selezionendo ripetutamente un vertice traboccante u e quindi "scaricandolo", cioè eseguendo operazioni di invio e di innalzamento finché u non ha più eccesso positivo. Quando un vertice viene innalzato, esso viene spostato in testa alla lista e l'algoritmo ricomincia a scandire la lista.

2. FIFO. Esamina i nodi attivi nell'ordine *first in first out* ed ha un tempo di esecuzione dell'ordine di $O(|V|^3)$. L'algoritmo FIFO mantiene un insieme gestendolo come una coda; seleziona il nodo u in testa alla coda, lo esamina, e aggiunge alla fine della lista gli eventuali nodi attivi formati. L'algoritmo studia il nodo finché è attivo oppure finché non è innalzato. In quest'ultimo caso si pone il nodo u alla fine della lista. L'algoritmo termina quando la coda dei nodi attivi è vuota.

3. MASSIMA ALTEZZA. Invia sempre flusso da un nodo ad un'altezza massima ed ha un tempo di esecuzione di $O(|V|^2\sqrt{|E|})$ che è migliore del precedente nel caso di grafi sparsi.

4. SCALING. L'idea intuitiva dell'algoritmo SCALING ideato da Ahuja e Orlin [2] è quella di inviare, quando possibile, un'elevata quantità di flusso. L'algoritmo utilizza un limite di eccesso Δ e un fattore scaling intero $k \geqslant 2$. Diremo che un vertice u ha un *eccesso elevato* se

$e(u) > \Delta/k$, altrimenti avrà un *eccesso basso*. Inizialmente Δ è la più piccola potenza di k tale che $\Delta \geqslant U$, dove U è un limite superiore per le capacità nella rete di flusso, inoltre durante l'esecuzione dell'algoritmo k rimane costante mentre Δ diminuisce. L'algoritmo mantiene l'invariante che $e(u) \leqslant \Delta$ per ogni vertice attivo u. Questa richiesta muta l'operazione di invio, che si applica se u è traboccante e se (u, v) è ammissibile, nel seguente modo:

PUSH-SCALING(u, v)

1. Se $v \neq t$ allora
2. $\quad f(u, v) = f(u, v) + \min\{e(u), c_f(u, v), \Delta - e(v)\}$
3. altrimenti
4. $\quad f(u, v) = f(u, v) + \min\{e(u), c_f(u, v)\}$

L'algoritmo consiste di un numero di *fasi scaling* durante ognuna delle quali Δ rimane costante. Ogni fase consiste nel ripetere i passi di PUSH-SCALING/LIFT finché non esiste più un vertice traboccante con eccesso elevato, e nel sostituire Δ con Δ/k. L'algoritmo termina quando non ci sono più vertici attivi. Inoltre tra tutti i vertici di eccesso elevato viene selezionato quello con altezza minima. In questo modo se le capacità sono intere l'algoritmo termina dopo al più $\lfloor \log_k U + 1 \rfloor$ fasi. Dopo questo numero di fasi risulta $\Delta < 1$ il che implica che f è un flusso. Il numero totale di invii non saturanti per l'algoritmo SCALING è $O(k|V|^2(\log_k U + 1))$.

5. STACK-SCALING. Questo algoritmo si basa sull'algoritmo precedente cercando però di migliorare il tempo di esecuzione sfruttando un fattore scaling k non costante e utilizzando la procedura STACK-PUSH/LIFT che svolge una sequenza di invii e innalzamenti su un vertice u traboccante sfruttando una pila. Anche l'algoritmo STACK-SCALING è costituito da fasi ognuna delle quali consiste nella ripetizione del passo STACK-PUSH/LIFT per un vertice attivo con eccesso elevato e altezza massima;

una fase termina quando non ci sono più vertici attivi con eccesso elevato. Si ottiene che il numero totale di invii non saturanti per l'algoritmo STACK-SCALING è $O(k|V|^2 + |V|^2(\log_k U + 1))$. Come per gli algoritmi di Goldberg-Tarjan e Ahuja-Orlin il tempo per effettuare operazioni di invio saturante ed innalzamento, e per esaminare archi ammissibili è $O(|V||E|)$. L'unico punto rimasto irrisolto è come scegliere i vertici da aggiungere alla pila. A tal fine si mantiene una struttura dati consistente di una collezione di liste:

$$list(j) = \{i \in N \mid e(i) > \Delta/k, h(i) = j\} \quad \forall j \in \{1, 2, 3, \ldots, 2|V| - 1\}.$$

Manterremo inoltre un puntatore per ricordare il più grande j per cui la rispettiva lista non è vuota. Ogni cambiamento del puntatore è dovuto ad un innalzamento oppure ad un cambiamento di fase. Quindi il numero di volte che il puntatore ha bisogno di essere aumentato o decrementato è dell'ordine di $O(|V|^2 + |V|(\log_k U + 1))$. La complessità globale dell'algoritmo è quindi $O(|V||E| + k|V|^2 + |V|^2(\log_k U + 1))$. Scegliendo $k = \lceil \log U / \log \log U \rceil$ si ha il seguente risultato.

Teorema. *L'algoritmo* STACK-SCALING*, con una opportuna scelta di k, ha una complessità dell'ordine di $O(|V||E| + |V|^2 \log U / \log \log U)$.*

6. ONDA-SCALING. Un altro modo per ridurre il numero degli invii non saturanti è di lavorare con l'*eccesso totale*, definito come la somma degli eccessi di tutti i vertici attivi. Il punto importante è che se l'eccesso totale è sufficientemente elevato, l'algoritmo può ottenere miglioramenti significativi applicando STACK-PUSH/LIFT ad ogni vertice attivo, selezionando i vertici in ordine topologico rispetto all'insieme degli archi ammissibili. L'algoritmo di *onda-scaling* sembra non avere alcun beneficio nell'utilizzare un fattore scaling non costante, quindi si fissa per comodità $k = 2$. L'algoritmo usa un altro parametro, $l \geqslant 1$. Una fase dell'algoritmo ONDA-SCALING consiste di due parti. La prima, il

passo onda, si ripete finché l'eccesso totale è minore di $|V|\Delta/l$. Quindi le operazioni di STACK-PUSH/LIFT sono applicate solo a vertici attivi di eccesso elevato in un qualche ordine, finché non ci sono più vertici traboccanti di eccesso elevato. Scegliendo $l = \sqrt{\log U}$ il tempo di esecuzione è $O(|V||E| + |V|^2\sqrt{\log U})$.

Gli approcci forniti fino a questo momento riducono il numero totale di invii. Un altro metodo per aumentare l'efficienza della soluzione del problema del massimo flusso su una rete è quello di ridurre il tempo totale per gli invii senza necessariamente ridurne il numero. Questo può essere fatto utilizzando la struttura dati di alberi dinamici di Sleator e Tarjan [11][2] studiata ampiamente per migliorare la complessità degli algoritmi sulle reti di flusso. La struttura dati di alberi dinamici mantiene una collezione di alberi radicati costituiti da vertici disgiunti e da archi dotati di un valore reale e permette al flusso di essere mosso attraverso un cammino piuttosto che attraverso un singolo arco alla volta. Analizziamo ora come gli alberi dinamici vengono utilizzati nello studio del flusso massimo in una rete di flusso. Diremo che, data una versione dell'algoritmo di preflusso con un limite di $p \geqslant |V||E|$ sul numero totale di invii, il tempo complessivo di esecuzione può essere ridotto da $O(p)$ a $O(|V||E| \log((p/|V||E|) + 1))$ usando gli alberi dinamici. Per esempio utilizzando gli alberi dinamici e la regola di selezione FIFO l'algoritmo di Goldberg e Tarjan ha un limite di $O(|V||E| \log(|V|^2/|E|))$. La stessa idea, applicata all'algoritmo di ONDA-SCALING, riduce il tempo di esecuzione a $O(|V||E| \log((|V|/|E|)\sqrt{\log U} + 2))$ e, sull'algoritmo di STACK-SCALING, dà un limite di $O(|V||E| \log(|V| \log U/|E| \log \log U) + 2))$. Abbiamo quindi analizzato l'algoritmo TREE-ONDA-SCALING, che si basa sulle le precedenti considerazioni ed ha un tempo di esecuzione dell'ordine di $O(|V||E| \log((|V|/|E|)\sqrt{\log U} + 2))$.

In conclusione la nostra tesi è suddivisa nel seguente modo: nel primo capitolo vengono presentate le principali definizioni, viene analizzato il teorema "Flusso Massimo Taglio Minimo" e vengono affrontati i primi due algoritmi che sfruttano il concetto di cammino aumentante, l'algoritmo FORD-FULKERSON e l'algoritmo EDMONDS-KARP; nel secondo capitolo siamo quindi entrati nel vivo nella nostra discussione studiando nel dettaglio l'algoritmo generico di preflusso proposto da Goldberg; nel terzo capitolo abbiamo analizzato tre algoritmi di preflusso che selezionano in modo strategico i vertici da cui effettuare l'operazione di invio, LIFT-TO-FRONT, FIFO e MASSIMA ALTEZZA; nel quarto capitolo abbiamo analizzato una variante dell'algoritmo di preflusso, dovuta ad Ahuja e Orlin che, utilizzando un cambio di dimensione chiamato "scaling", risolve il problema in un tempo di $O(|V||E|+|V|^2 \log U)$ su una rete con capacità intere limitate da U e abbiamo visto come l'inserimento di una opportuna struttura dati, come quella di alberi dinamici studiata da Sleator e Tarjan, comporta un algoritmo più complicato, rispetto a quello proposto da Goldberg, ma con un tempo di esecuzione di $O(|V||E| \log(|V|^2/|E|))$.

Oltre alla trattazione teorica sono riportati i listati di tre programmi. Si tratta delle implementazioni in linguaggio C di tre degli algoritmi descritti in questa tesi: l'algoritmo di preflusso LIFT-TO-FRONT, l'algoritmo STACK-SCALING e quello più complesso che utilizza la struttura di alberi dinamici di Tarjan e Sleator TREE-ONDA-SCALING.

Bibliografia

[1] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, 1993, pp. 167-273.

[2] R.K. Ahuja, R.E. Tarjan, J.B. Orlin, *Improved time bounds for the maximum flow problem*, SIAM J. Comput. 18, 1989, pp. 939-954.

[3] E. Althaus, K. Mehlhorn, *Maximum network flow with floating point arithmetic*, Information Processing Letters 66, 1998, pp 109-113.

[4] H. Booth, R.E. Tarjan, *Finding the minimum cost maximum flow in a series-parallel network*, Journal of Algorithms 15, 1993, pp. 416-446.

[5] J. Cheriyan, T. Hagerup, K. Mehlhorn, *An $O(n^3)$-time Maximum flow algorithm*, SIAM J. Comput. 25, 1996, pp. 1144-1170.

[6] T.H. Cormen,C.E. Leiserson,R.L. Rivest, *Introduzione agli algoritmi(vol.2)*, Jackson Libri, 1994.

[7] Dexter, C. Kozen, *The design and analysis of algorithms*, 1993, pp 84-100.

[8] L.R. Ford, D.R. Fulkerson, *Flows in Networks*, Princeton University Press,1962.

[9] H.N. Gabow, R.E. Tarjan, *Faster scaling algorithms for network problems*, SIAM J. Comput. 5, 1989, pp. 1013-1036.

[10] G. Gallo, M.D. Grigoriadis, R.E. Tarjan, *A fast parametric meximum flow algorithm and applications*, SIAM J. Comput. 18, 1989, pp. 30-55.

[11] A.V. Goldberg, M.D. Grigoriadis, R.E. Tarjan, *Use of dinamic trees in a network simplex algorithm for the maximum flow problem*, Mathematical Programming 50, 1991, pp. 277-290.

[12] A.V. Goldberg,R.E. Tarjan, *A new approach to the maximum flow problem*, Proc. 18th Ann. ACM Symp. on Theory of Computing, Berkeley, 1986, pp. 136-146.

[13] J.V. Leeuwen, *Handbook of Theoretical Computer Science*, 1990, pp. 589-603.

[14] R.E. Tarjan, *Data structures and network algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics 44, 1983, cap. 8, pp. 97-110.

[15] R.E. Tarjan, S. Rao, V. King *A faster deterministic maximum flow algorithm*, Journal of algorithms 17, 1994, pp. 447-474.