TECNICA *GREEDY*

1. Introduzione

Un algoritmo greedy tenta di costruire una soluzione ottima partendo da una soluzione parziale iniziale ed estendendola a poco a poco fino a quando questo non è più possibile. Quando l'algoritmo tenta di estendere una *soluzione parziale* non lo fa considerando tutte le possibili estensioni (che potrebbero essere numerosissime) ma solamente quelle che chiamiamo *estensioni locali*. Le estensioni locali di una soluzione parziale rappresentano in qualche modo le più piccole estensioni possibili e sono relativamente poche. Fra tutte le estensioni locali l'algoritmo sceglie la *più conveniente*. Ovvero quella estensione che sembra, almeno localmente, la più promettente per raggiungere una soluzione ottima. Uno schema generale per gli algoritmi greedy è il seguente:

INPUT I (Istanza del problema)
  $S \leftarrow S_0$ (Soluzione parziale iniziale per I)
  WHILE S può essere estesa DO
    Trova l'estensione locale S' di S più conveniente
    $S \leftarrow S'$
  ENDWHILE
OUTPUT S

Ovviamente questo schema non è una ricetta per costruire algoritmi greedy, ma è solamente una descrizione di ciò che si intende per algoritmo greedy. È chiaro infatti che ci sono molti dettagli, tutt'altro che marginali, che lo schema non tenta di specificare:

> Che cosa si intende per soluzione parziale? Vale a dire, dato il problema che si vuole risolvere come sono definite le soluzioni parziali? Come deve essere scelta la soluzione parziale iniziale? Quali sono le estensioni locali di una soluzione parziale? Qual'è il criterio che definisce la convenienza di una estensione locale?

Il fatto è che per un dato problema ci possono essere molti algoritmi che possiamo considerare greedy. Questo lo si può vedere bene tramite alcuni esempi.

Consideriamo il seguente problema chiamato SELEZIONE ATTIVITÀ. Date n attività (lezioni, seminari, ecc.) e per ogni attività i, $1 \leq i \leq n$, l'intervallo temporale $[s_i, f_i)$ in cui l'attività dovrebbe svolgersi, selezionare il maggior numero di attività che possono essere svolte senza sovrapposizioni in un'unica aula. Ad esempio se si hanno le seguenti 5 attività:

1	2	3	4	5
$[5,7)$	$[2,5)$	$[3,4)$	$[8,10)$	$[9,10)$

allora se ne possono selezionare 3 che possono essere svolte senza sovrapposizioni (ad esempio, le attività $1, 2, 4$ oppure $1, 3, 5$).
Per questo problema è naturale considerare come soluzione parziale un qualsiasi insieme di attività che non si sovrappongono. Si osservi che considerare come soluzione parziale un qualsiasi insieme di attività (che possono anche sovrapporsi) non è in linea con la filosofia degli algoritmi greedy. Infatti, una soluzione parziale dovrebbe essere sempre estesa ma mai ridotta. Vale a dire, non si torna mai su una decisione che è stata già presa.
Anche il concetto di estensione locale è naturale, semplicemente, l'aggiunta di una attività che non si sovrappone a quelle già presenti nella soluzione parziale. Invece, il criterio per misurare la convenienza di una estensione locale non è così facile da scegliere. Un possibile criterio potrebbe consistere nel preferire, tra tutte le attività che possiamo aggiungere, quella (o una di quelle) che inizia prima. Oppure quella che dura meno; oppure quella che termina prima. Quindi potremmo proporre almeno i seguenti tre algoritmi greedy per il problema SELEZIONE ATTIVITÀ:

Algoritmo INIZIA_PRIMA
INPUT Gli intervalli temporali di n attività $[s_1, f_1), [s_2, f_2), \ldots, [sn, fn)$
  $SOL \leftarrow \emptyset$
  WHILE esistono attività che possono essere aggiunte a SOL senza sovrapposizioni DO
    Sia k una attività con minimo tempo d'inizio fra quelle che possono essere aggiunte a SOL

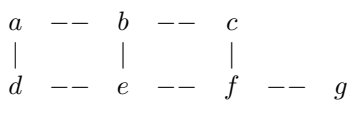
$SOL \leftarrow SOL \cup \{k\}$
ENDWHILE
OUTPUT SOL.

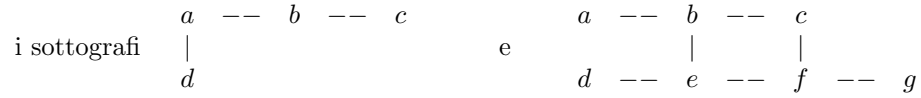
Algoritmo DURA_MENO
INPUT Gli intervalli temporali di n attività $[s_1, f_1), [s_2, f_2), \ldots, [sn, fn)$
$SOL \leftarrow \emptyset$
WHILE esistono attività che possono essere aggiunte a SOL senza sovrapposizioni DO
Sia k una attività di minima durata fra quelle che possono essere aggiunte a SOL
$SOL \leftarrow SOL \cup \{k\}$
ENDWHILE
OUTPUT SOL.

Algoritmo TERMINA_PRIMA
INPUT Gli intervalli temporali di n attività $[s_1, f_1), [s_2, f_2), \ldots, [sn, fn)$
$SOL \leftarrow \emptyset$
WHILE esistono attività che possono essere aggiunte a SOL senza sovrapposizioni DO
Sia k una attività con minimo tempo di fine fra quelle che possono essere aggiunte a SOL
$SOL \leftarrow SOL \cup \{k\}$
ENDWHILE
OUTPUT SOL.

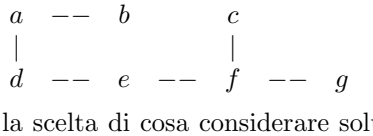
Discuteremo dopo se e quali di questi algoritmi trovano sempre una soluzione ottima per il problema.

Adesso consideriamo un'altro problema chiamato MINIMO ALBERO DI COPERTURA. Dato un grafo non diretto e connesso $G = (V, E)$ con archi pesati, trovare un albero di copertura di peso minimo. Per peso di un albero si intende la somma dei pesi degli archi dell'albero. Si ricorda che un sottografo è detto coprente o di copertura se ogni nodo del grafo è contenuto nel sottografo. Ad esempio per il grafo qui sotto disegnato:





non sono alberi di copertura. Il primo perché pur essendo un albero non copre tutti i nodi del grafo e il secondo perché pur coprendo tutti i nodi non è un albero. Invece quello qui sotto disegnato è un albero di copertura:



Per il MINIMO ALBERO DI COPERTURA la scelta di cosa considerare soluzione parziale non è così chiara come per il problema precedente. In effetti vengono in mente almeno due possibilità. L'albero potrebbe essere costruito a partire da un nodo (scelto in modo arbitrario) e aggiungendo via via archi che estendono l'albero. Così una soluzione parziale è un albero (sottografo del grafo di input) che è radicato nel nodo scelto. Un'altra possibilità sta nel considerare come soluzione parziale un qualsiasi sottoinsieme degli archi che non contiene cicli. In questo caso gli archi che via via si aggiungono "connettono" sempre più il sottografo fino ad arrivare a un sottografo connesso ed aciclico, cioè, un albero. In entrambi i casi l'estensione locale consiste nell'aggiunta di un arco. Nel primo caso un arco che estende l'albero e nel secondo un arco che non crea cicli. Anche il criterio che definisce la scelta più conveniente è uguale nei due casi:l'arco di peso minimo. Quindi si possono proporre i seguenti due algoritmi greedy per il problema:

Algoritmo PRIM
INPUT Un grafo non diretto e connesso $G = (V, E)$ con archi pesati
    $SOL \leftarrow \emptyset$
    Scegli un nodo s di G
    $C \leftarrow \{s\}$
    WHILE $C \neq V$ DO (Finché l'albero non copre tutti i nodi)
        Sia $\{u, v\}$ un arco di peso minimo fra tutti quelli con u in C e v non in C
        $SOL \leftarrow SOL \cup \{\{u, v\}\}$
        $C \leftarrow C \cup \{v\}$
    ENDWHILE
OUTPUT SOL.

Algoritmo KRUSKAL
INPUT Un grafo non diretto e connesso $G = (V, E)$ con archi pesati
    $SOL \leftarrow \emptyset$
    WHILE esiste un arco in E che può essere aggiunto a SOL senza creare cicli DO
        Sia $\{u, v\}$ un arco di peso minimo fra tutti quelli che possono essere aggiunti a SOL
        $SOL \leftarrow SOL \cup \{\{u, v\}\}$
    ENDWHILE
OUTPUT SOL.

I nomi dei due algoritmi provengono dai nomi dei due ricercatori che per primi li hanno studiati e resi pubblici.

A questo punto la trattazione procede considerando singoli problemi e analizzando possibili algoritmi greedy per tali problemi. L'analisi cercherà in primo luogo di determinare se l'algoritmo è *corretto*, cioè, trova sempre una soluzione ottima e in secondo luogo saranno discusse possibili implementazioni *efficienti*.

2. Il Problema SELEZIONE ATTIVITÀ e uno Schema Generale per Dimostrare la Correttezza degli Algoritmi Greedy

Consideriamo il problema SELEZIONE ATTIVITÀ precedentemente definito. Vogliamo analizzare i tre algoritmi proposti. Prima di avventurarsi in un tentativo di dimostrazione di correttezza di un algoritmo conviene pensare a qualche semplice istanza del problema che potrebbe mettere in difficoltà l'algoritmo. Nell'algoritmo INIZIA_PRIMA le attività sono scelte in ordine di tempo d'inizio. Quindi se c'è un'attività che inizia prima di altre con le quali è incompatibile mentre queste sono tra loro compatibili allora la scelta di tale attività può pregiudicare il raggiungimento di una soluzione ottima. Infatti, consideriamo la seguente semplice istanza:

1	2	3
$[1, 10)$	$[2, 3)$	$[4, 5)$

Chiaramente, l'algoritmo fornirebbe la soluzione composta dalla sola attività 1 mentre la soluzione ottima ha due attività (2 e 3). Quindi l'algoritmo non è corretto.

Nell'algoritmo DURA_MENO le attività sono scelte in ordine di durata. Se applicato all'istanza precedente in effetti l'algoritmo trova la soluzione ottima. Però potrebbe esserci un'attività con una durata piccola che è incompatibile con due attività di durata maggiore che sono tra loro compatibili. La scelta dell'attività piccola può pregiudicare il raggiungimento dell'ottimo. Infatti, consideriamo l'istanza:

1	2	3
$[5, 7)$	$[3, 6)$	$[6, 9)$

Chiaramente, l'algoritmo fornisce la soluzione composta dalla sola attività 1 mentre la soluzione ottima ha due attività (2 e 3). Quindi l'algoritmo non è corretto.

2.1. **Correttezza dell'Algoritmo TERMINA_PRIMA e lo Schema Generale.** L'algoritmo TERMINA_PRIMA produce soluzioni ottime per entrambe le istanze che mettono in difficoltà gli altri due algoritmi. È corretto? Se non si riesce a trovare un'istanza che mette in difficoltà un algoritmo, cioè, non si riesce a trovare un *controesempio* alla correttezza dell'algoritmo, allora si può tentare di dimostrarne la correttezza. Il tentativo potrebbe fallire e il modo in cui fallisce potrebbe darci indicazioni su come trovare un controesempio. Ma a volte il tentativo fallisce nonostante l'algoritmo sia corretto. Semplicemente, potrebbe essere assai arduo trovare una dimostrazione di correttezza. Non ci sono ricette o procedure meccaniche che permettano di dirimere questioni di tal sorta. Spesso trovare una dimostrazione (non banale) richiede ingegnosità, creatività e fantasia. Le dimostrazioni di correttezza di algoritmi non fanno eccezione. Però nel caso di algoritmi greedy si possono dare delle linee guida che spesso risultano molto utili nella costruzione di una dimostrazione di correttezza per tali algoritmi. Queste linee guida forniscono uno *schema generale* che decompone l'ipotetica dimostrazione in parti abbastanza piccole e ben delineate così da facilitarne l'analisi. Lo schema generale è il seguente. Prima di tutto si cerca di dimostrare che ogni soluzione parziale prodotta ad un certo passo dell'algoritmo è estendibile ad una opportuna soluzione ottima. Ciò equivale a dimostrare che le decisioni prese dall'algoritmo non pregiudicano mai l'ottenimento di una soluzione ottima (non prende mai decisioni "sbagliate"). Dopo aver dimostrato ciò si sa per certo che la soluzione finale prodotta dall'algoritmo è estendibile ad una soluzione ottima. Rimane quindi da dimostrare che la soluzione finale è invero uguale a tale soluzione ottima. Ciò equivale a dimostrare che l'algoritmo prende un numero sufficiente di decisioni "giuste". Altre osservazioni sullo schema generale saranno presentate dopo aver visto tale schema applicato all'algoritmo TERMINA_PRIMA.

Prima di tutto è necessario dimostrare che l'algoritmo effettua sempre un numero finito di passi. Nel caso dell'algoritmo TERMINA_PRIMA è facile. Infatti, è sufficiente osservare che ad ogni iterazione del WHILE viene aggiunta una nuova attività a SOL e la condizione del WHILE diventa falsa non appena non ci sono più attività che possono essere aggiunte a SOL. Quindi il WHILE può eseguire al più n iterazioni. Indichiamo con SOL_h la soluzione parziale prodotta durante l'h-esima iterazione del WHILE e sia SOL_0 la soluzione parziale iniziale. Sia m il numero di iterazioni eseguite dal WHILE. Nel caso dell'algoritmo TERMINA_PRIMA dimostrare che ogni soluzione parziale prodotta è estendibile ad una soluzione ottima significa dimostrare la seguente affermazione:

Per ogni $h = 0, 1, \ldots, m$ esiste una soluzione ottima SOL^* che contiene SOL_h.

Dimostrazione La dimostrazione procede per induzione su h. Per $h = 0$ l'asserto è banalmente vero in quanto $SOL_0 = \emptyset$. Sia ora $h \geq 0$, vogliamo dimostrare che se l'asserto è vero per h allora è anche vero per $h + 1$ (sempreché $h + 1 \leq m$). L'ipotesi induttiva ci dice quindi che esiste una soluzione ottima SOL^* che contiene SOL_h. Se SOL^* contiene anche SOL_{h+1} allora abbiamo fatto. Se invece SOL^* non contiene SOL_{h+1} allora deve essere che l'attività k aggiunta a SOL_h nell'$(h + 1)$-esima iterazione non appartiene a SOL^*. Ora vogliamo far vedere che è possibile trasformare SOL^* in un'altra soluzione ottima che invece contiene SOL_{h+1}. Quindi tale soluzione ottima dovrà contenere l'attività k (oltre a tutte le attività di SOL_h) e siccome non può superare la cardinalità di SOL^* non potrà contenere qualche attività che è in SOL^*. Dobbiamo trovare una attività in SOL^* che può essere sostituita dall'attività k. Osserviamo che SOL_{h+1} è una soluzione ammissibile, cioè, le attività in SOL_{h+1} sono tutte compatibili fra loro. Infatti SOL_h è ammissibile perché è contenuta in una soluzione ottima e l'attività k è una attività che può essere aggiunta a $SOLh$, cioè, è compatibile con tutte le attività in SOL_h. Quindi, $|SOL^*| \geq |SOL_{h+1}|$ e allora deve esistere almeno una attività in SOL^* che non appartiene a SOL_{h+1}. Sia j quella fra queste attività che termina per prima. Definiamo $SOL^{\#} = (SOL^* - \{j\}) \cup \{k\}$. Dobbiamo far vedere che $SOL^{\#}$ è una soluzione ammissibile. Siccome j appartiene a SOL^* e non appartiene a SOL_h, allora j era fra le attività che potevano essere aggiunte a SOL_h nell'$(h + 1)$-esima iterazione. Dato che è stata scelta l'attività k, deve essere $f_k \leq f_j$. Allora non ci sono altre attività in SOL^* oltre a j che sono incompatibili con k. Infatti, tutte le attività in $(SOL^* - \{j\}) - SOL_{h+1}$ terminano dopo j e sono compatibili con j, per cui iniziano dopo j e quindi anche dopo k. Dunque $SOL^{\#}$ è una soluzione ammissibile con la stessa cardinalità di una soluzione ottima, quindi è una soluzione ottima. Inoltre, $SOL^{\#}$ contiene SOL_{h+1} e questo completa la dimostrazione.

A questo punto sappiamo che la soluzione finale SOL_m prodotta dall'algoritmo è contenuta in una soluzione ottima SOL^*. Supponiamo per assurdo che SOL_m sia differente da SOL^*. Allora, esiste almeno una attività j in SOL^* che non appartiene a SOL_m. Siccome l'attività j è compatibile con tutte le attività in SOL_m e non appartiene a SOL_m la condizione del WHILE all'inizio della $(m + 1)$-esima iterazione è vera in contraddizione con il fatto che il WHILE esegue solamente m iterazioni. Dunque deve essere $SOL_m = SOL^*$. Abbiamo così dimostrato che l'algoritmo TERMINA_PRIMA è corretto per il problema SELEZIONE ATTIVITÀ.

Le dimostrazioni come questa che seguono lo schema generale condividono alcuni altri aspetti oltre a quelli già menzionati. Prima di tutto, la dimostrazione che ogni soluzione parziale è estendibile a una soluzione ottima può sempre essere fatta per induzione sulle iterazioni dell'algoritmo. Anzi, l'enunciato da dimostrare ha proprio quella forma per facilitare una dimostrazione per induzione. Infatti, ciò che interessa, come si vede dalla dimostrazione precedente, è dimostrare che la soluzione finale prodotta dall'algoritmo è estendibile ad una soluzione ottima. Un'altro aspetto che tali dimostrazioni condividono è che nel corso della dimostrazione per induzione occorre trasformare la soluzione ottima, che per ipotesi induttiva estende la soluzione parziale di una certa iterazione, in un'altra soluzione ottima che estende la soluzione parziale dell'iterazione successiva. Spesso tale trasformazione consiste in una sostituzione di un opportuno elemento della soluzione ottima con un elemento della soluzione parziale. Questo è certo l'aspetto cruciale di queste dimostrazioni. Infatti, è soprattutto nella determinazione di tale trasformazione che le caratteristiche specifiche dell'algoritmo entrano in gioco.

2.2. **Implementazioni ed Algoritmi Efficienti.** Dopo aver dimostrato che un algoritmo è corretto ha interesse vedere se e come lo si può implementare in modo efficiente. Un algoritmo corretto ma non efficiente ha di solito scarso interesse. D'altronde per quasi tutti i problemi che si ha interesse risolvere esistono sempre algoritmi corretti. Ma solamente alcuni ammettono algoritmi anche efficienti.

Abbiamo parlato di implementazioni efficienti e di algoritmi efficienti basandoci sull'intuizione che dovrebbe essere stata maturata durante i corsi di programmazione e durante un primo corso di algoritmi. Ma qual'è la differenza fra implementazione ed algoritmo? Sono sinonimi? Che cosa si intende per implementazione efficiente? E per algoritmo efficiente? È venuto il momento chiarire il significato che conveniamo di dare a questi termini.

Per algoritmo intendiamo una descrizione non ambigua di una procedura che è sufficientemente precisa da essere compresa e possibilimente eseguita da un essere umano. Ciò significa che tutti quei dettagli che sarebbero indispensabili affinché la procedura possa essere codificata in un linguaggio di programmazione imperativo non devono necessariamente essere specificati in un algoritmo. Anzi è meglio che dettagli non necessari non siano specificati. In realtà si può dire di più. Un algoritmo dovrebbe descrivere soltanto quei passi e quelle decisioni che sono essenziali al raggiungimento dell'obbiettivo dell'algoritmo. Ad esempio l'algoritmo TERMINA_PRIMA non specifica quale attività scegliere se ve ne più d'una fra quelle compatibili con quelle già scelte con tempo di terminazione minimo. Ai fini del raggiungimento dell'obbiettivo una qualsiasi di queste attività va bene. O meglio chi ha pensato l'algoritmo crede o ha dimostrato che in questo caso scegliere l'una piuttosto che l'altra è irrilevante.

Per implementazione intendiamo invece una descrizione di una procedura che sia sufficientemente precisa da essere **direttamente** codificabile in un linguaggio di programmazione imperativo. Ciò significa che una implementazione, se non è già scritta in un linguaggio di programmazione, deve comunque essere così dettagliata da non lasciare dubbi o possibilità di scelta su come può essere codificata in un linguaggio di programmazione (ad esempio in linguaggio C). In generale un algoritmo può ammettere più implementazioni. Ciò è principalmente dovuto al fatto che la descrizione data dall'algoritmo si concentra sul cosa fare piuttosto che sul come farlo. Quindi, diverse implementazioni potrebbero basarsi su diversi modi di fare una stessa cosa. Ad esempio nel caso dell'algoritmo TERMINA_PRIMA, non viene specificato come trovare una attività di minimo tempo di terminazione fra quelle compatibili con quelle già scelte. Una implementazione deve invece specificare come trovare una tale attività.

Da questa discussione dovrebbe essere chiaro che non c'è una distinzione netta fra algoritmo ed implementazione. Infatti, ogni implementazione potrebbe anche essere chiamata algoritmo. Però non ogni algoritmo può essere chiamato anche implementazione, perché potrebbe non essere sufficientemente dettagliato. In ogni caso però, anche se non viene esplicitamente detto, se una implementazione viene chiamata algoritmo con questo termine ci si riferisce all'ipotetico algoritmo che soggiace all'implementazione. Cioè, una descrizione della procedura implementata che descrive nulla di più di cosa la procedura fa (le decisioni che la procedura prende) prescindendo quindi da dettagli implementativi.

Ma cosa si intende per *efficiente*? Non esiste una definizione rigorosa di questo concetto. Una discussione che potrebbe dirsi esauriente di questo aspetto richiederebbe competenze e spazio che esulano da questo corso. Però si possono indicare due proprietà che se possedute da un algoritmo lo fanno considerare efficiente. La prima proprietà è che esista una implementazione dell'algoritmo il cui tempo di esecuzione cresca lentamente al crescere della dimensione dell'istanza. Generalmente, questo significa che la complessità asintotica dell'implementazione è limitata da un polinomio di basso grado nella dimensione dell'istanza di input, ad esempio $O(n)$, $O(n^2)$ dove n è la dimensione dell'istanza. Già $O(n^3)$ potrebbe essere considerato un limite troppo elevato, ma questo può dipendere fortemente dal tipo di problema e dalle relative applicazioni. La seconda proprietà è più sottile e molto più difficile da verificare. Si deve esibire una implementazione dell'algoritmo di complessità asintotica vicina alla migliore possibile per il problema risolto dall'algoritmo. Ad esempio se il problema è la ricerca in un vettore ordinato, un algoritmo di complessità $O(n)$ non è considerato efficiente, nonostante soddisfi la prima proprietà. Come è noto, infatti, esiste un algoritmo di complessità $O(\log n)$ per tale problema. D'altronde il soddisfacimento della seconda proprietà non garantisce che l'algoritmo possa considerarsi efficiente. Basta pensare ad un problema per cui si sa

che un qualsiasi algoritmo deve avere complessità $O(2^n)$. Allora un algoritmo per questo problema anche se ha la migliore complessità possibile, cioè $O(2^n)$, non può essere considerato efficiente.

2.3. **Implementazione dell'Algoritmo TERMINA_PRIMA.** Le descrizioni in pseudo-codice che abbiamo visto finora hanno una forma atta a facilitare il più possibile una eventuale dimostrazione di correttezza. Per questo è conveniente una descrizione che sia precisa, non ambigua, ma allo stesso tempo essenziale. Tutti quei dettagli che sono propri di una implementazione renderebbero solamente più laboriosa una eventuale dimostrazione di correttezza. È chiaro però che anche da una descrizione così ad alto livello è immediato avere un'idea dell'efficienza di possibili implementazioni. Altrimenti sarebbe quasi inutile spendere tempo a tentare di dimostrare la correttezza di un algoritmo di cui non si ha la minima idea circa possibili implementazioni efficienti.

Quindi, partendo dalla descrizione in pseudo-codice è necessario determinare tutti quei dettagli che permettano di precisare una implementazione fino al punto di poterne effettuare una analisi asintotica della complessità. Questo significa che a tal fine non è necessario fornire una implementazione in un qualche linguaggio di programmazione ma è sufficiente una descrizione in pseudo-codice abbastanza dettagliata.

Tornando all'algoritmo TERMINA_PRIMA possiamo supporre innanzitutto che l'input sia rappresentato da un vettore i cui elementi sono le coppie tempo d'inizio, tempo di fine delle attività. Ad ogni iterazione occorre scegliere l'attività con il minimo tempo di fine tra quelle compatibili, per rendere efficiente ciò conviene ordinare preliminarmente le attività per tempi di fine non decrescenti. Fatto questo, per determinare la compatibilità è sufficiente controllare che il tempo d'inizio della prossima attività non preceda il tempo di fine dell'ultima attività selezionata. La descrizione in pseudo-codice di una tale implementazione è la seguente:

```
INPUT Un vettore A[1...n] tale che A[i].s e A[i].f sono i tempi d'inizio e di fine dell'attività i
    SOL ← ∅            (l'insieme SOL può essere implementato tramite una lista)
    Calcola un vettore P che ordina il vettore A rispetto ai tempi di fine, cioè,
    A[P[1]].f ≤ A[P[2]].f ≤ ... ≤ A[P[n]].f
    t ← 0              (si assume che i tempi d'inizio sono non negativi)
    FOR j ← 1 TO n DO
        IF A[P[j]].s ≥ t THEN
            SOL ← SOL ∪ {P[j]}
            t ← A[P[j]].f
        ENDIF
    ENDFOR
OUTPUT SOL.
```

L'ordinamento si può fare in $O(n \log n)$ usando uno degli algoritmi classici (ad esempio merge-sort). Ciascuna delle n iterazioni del FOR ha costo costante. Quindi, questa implementazione dell'algoritmo TERMINA_PRIMA ha complessità asintotica $O(n \log n)$, dove n è il numero di attività.

3. Il Problema MINIMO ALBERO DI COPERTURA

3.1. **Correttezza dell'Algoritmo PRIM.** Abbiamo già visto due algoritmi greedy per il problema MINIMO ALBERO DI COPERTURA, adesso li analizzeremo. Iniziamo dall'algoritmo PRIM. Dimostremo che è corretto seguendo lo schema generale. Riportiamo per comodità l'algoritmo qui sotto:

```
INPUT Un grafo non diretto e connesso G = (V, E) con archi pesati
    SOL ← ∅
    Scegli un nodo s di G
    C ← {s}
    WHILE C ≠ V DO          (Finché l'albero non copre tutti i nodi)
        Sia {u, v} un arco di peso minimo fra tutti quelli con u in C e v non in C
        SOL ← SOL ∪ {{u, v}}
        C ← C ∪ {v}
    ENDWHILE
OUTPUT SOL.
```

La primissima cosa da dimostrare è che tutti i passi dell'algoritmo possono essere effettivamente eseguiti. Nel caso dell'algoritmo TERMINA_PRIMA non abbiamo esplicitamente considerato ciò perché era evidente. Ma in questo caso il passo:

Sia $\{u, v\}$ un arco di peso minimo fra tutti quelli con u in C e v non in C

non è evidente che possa sempre essere eseguito, cioè, non è immediatamente evidente che esista sempre almeno un arco con un nodo in C e l'altro non in C. Occorre quindi mostrare che se la condizione del WHILE e vera, cioè $C \neq V$, allora almeno un arco con tale proprietà esiste. Infatti, se $C \neq V$ allora c'è almeno un nodo w che non è in C. Siccome il grafo G è supposto essere connesso esiste un cammino da s a w. Questo cammino parte dal nodo s in C e arriva al nodo w non in C. Se si percorre il cammino a partire da s fino a che si incontra il primo nodo y che non è in C, allora l'arco $\{x, y\}$, dove x è il nodo che precede y, ha la proprietà richiesta.

Ora dobbiamo mostrare che l'algoritmo termina sempre, cioè, il WHILE esegue sempre un numero finito di iterazioni. Ciò equivale a dimostrare che esiste sempre una iterazione in cui la condizione del WHILE risulta falsa. Vale a dire una iterazione in cui risulta $C = V$. Ad ogni iterazione del WHILE viene aggiunto un nuovo nodo a C, quindi vengono eseguite esattamente $|V|$–1 iterazioni.

Sia SOL_h il valore di SOL al termine dell'h-esima iterazione del WHILE e sia SOL_0 il valore iniziale. Proviamo che ogni soluzione parziale può essere estesa ad una soluzione ottima:

Per ogni $h = 0, 1, \ldots, |V|$–1, esiste una souzione ottima SOL^* che contiene SOL_h.

Dimostrazione. Per induzione su h. Per $h = 0$ si ha $SOL_0 = \emptyset$ quindi l'asserto è vero. Sia $h \geq 0$, vogliamo provare che se l'asserto è vero per h (ipotesi induttiva) allora è vero anche per $h + 1$ (sempre che $h + 1 \leq |V|$–1). Per ipotesi induttiva esiste una soluzione ottima SOL^* che contiene SOL_h. Se SOL^* contiene anche SOL_{h+1} allora abbiamo fatto. Altrimenti, l'arco $\{u, v\}$ aggiunto a SOL durante l'$(h + 1)$-esima iterazione non appartiene a SOL^*. Vogliamo trasformare SOL^* in un'altra soluzione ottima che però contiene SOL_{h+1}. Siccome SOL^* è un albero di copertura, l'arco $\{u, v\}$ determina un ciclo in $SOL^* \cup \{u, v\}$. Vogliamo far vedere che c'è un arco di questo ciclo che possiamo sostituire con l'arco $\{u, v\}$. L'arco che cerchiamo deve non appartenere a SOL_{h+1} e avere peso maggiore od uguale a quello di $\{u, v\}$. L'arco $\{u, v\}$ è stato scelto come un arco di peso minimo fra tutti gli archi con u in C e v non in C. Quindi se troviamo un'altro arco del ciclo con questa proprietà abbiamo fatto. Siano $x_1, x_2, \ldots, x_k$ i nodi del ciclo presi nell'ordine tale che $x_1 = u$ e $x_k = v$:

$$u = x_1 - - x_2 - - - \ldots - - x_{k-1} - - x_k = v$$

Partendo da x_1 percorriamo i nodi del ciclo in tale ordine fino a che troviamo il primo nodo che non appartiene a C. Sia x_j questo nodo, allora l'arco $\{x_{j-1}, x_j\}$ è un arco del ciclo diverso da $\{u, v\}$ tale che x_{j-1} è in C e x_j non è in C. Definiamo $SOL^{\#} = (SOL^* - \{\{x_{j-1}, x_j\}\}) \cup \{\{u, v\}\}$. L'arco $\{x_{j-1}, x_j\}$ ha un peso maggiore od uguale a quello dell'arco $\{u, v\}$, per cui $SOL^{\#}$ è una soluzione ottima che contiene SOL_{h+1} e questo completa la dimostrazione.

Ora sappiamo che esiste una soluzione ottima SOL^* che contiene la soluzione finale $SOL_{|V|-1}$ prodotta dall'algoritmo. Essendo SOL^* un albero di copertura deve avere esattamente $|V|$–1 archi. Siccome anche $SOL_{|V|-1}$ ha esattamente $|V|$–1 archi, deve essere che $SOL_{|V|-1}$ coincide con SOL^*. Dunque l'algoritmo di Prim è corretto per il problema MINIMO ALBERO DI COPERTURA.

3.2. **Implementazione dell'Algoritmo PRIM.** Consideriamo ora possibili implementazioni efficienti dell'algoritmo PRIM. Il passo critico su cui concentrarsi per trovare una implementazione efficiente è chiaramente il seguente:

Sia $\{u, v\}$ un arco di peso minimo fra tutti quelli con u in C e v non in C.

Ovvero, si tratta di trovare un modo conveniente per rappresentare ed organizzare le informazioni necessarie per eseguire tale passo efficientemente. Assumiamo che il grafo di input venga rappresentato tramite liste di adiacenza con pesi e che i nodi siano numerati a partire da 1. Inoltre, rappresentiamo l'albero di copertura tramite un vettore dei padri, cioè, un vettore T tale che $T[r] = r$ se r è la radice dell'albero e per ogni $x \neq r, T[x] = y$ dove y è il padre di x.

In analogia con l'implementazione dell'algoritmo TERMINA_PRIMA potremmo pensare di rendere più efficiente la scelta di un arco di peso minimo fra quelli che soddisfano la proprietà ordinando preliminarmente gli archi del grafo in ordine di peso non decrescente. Però, in questo caso, quando un arco non può essere aggiunto alla soluzione parziale perché viola la proprietà in una certa iterazione potrebbe venir aggiunto in una iterazione successiva. Quindi, l'ordinamento preliminare non risulta utile in quanto è necessario ad ogni iterazione riesaminare dall'inizio la sequenza degli archi. La complessità

asintotica sarebbe almeno dell'ordine di $O(|V||E|)$, cioè, ordine di $|V|$ iterazioni ognuna con costo dell'ordine di almeno $|E|$. Vorremmo invece trovare una implementazione più efficiente.

Osserviamo che ad ogni iterazione viene aggiunto un nuovo arco e anche un nuovo nodo all'albero. Se vediamo l'estensione dell'albero in termini di nodi aggiunti invece che di archi, qual'è il criterio per la scelta del nodo da aggiungere? Un nodo che estende l'albero di costo minimo fra tutti i costi dei nodi che possono estendere l'albero, dove per costo di un nodo si intende il peso minimo fra i pesi degli archi che connettono il nodo all'albero (se non ci sono archi che connettono il nodo all'albero si conviene che abbia costo infinito). In questa interpretazione dell'algoritmo i costi dei nodi, a differenza dei pesi degli archi, devono essere aggiornati ad ogni iterazione. Infatti, dopo che un nuovo nodo è stato aggiunto il costo di qualche adiacente al nuovo nodo può diminuire. Per mantenere i costi dei nodi possiamo usare un semplice vettore $COSTO$ tale che $COSTO[x]$ è il costo del nodo x. Dobbiamo anche mantenere l'insieme dei nodi finora coperti, per questo usiamo un vettore caratteristico C, cioè, tale che $C[x] = 1$ se il nodo x è coperto e $C[x] = 0$ altrimenti. La descrizione in pseudo-codice di questa implementazione è la seguente:

```
INPUT Un vettore A[1...n] di liste di adiacenza con pesi di un grafo non diretto, connesso e con archi
    pesati
    FOR i ← 1 TO n DO C[i] ← 0          (Inizialmente, nessun nodo è coperto)
    T[1] ← 1          (Scegli come radice il nodo 1)
    COSTO[1] ← 0
    FOR i ← 2 TO n DO COSTO[i] ← infinito
    REPEAT n TIMES
        Fra tutti i nodi non ancora coperti trova un nodo k di costo minimo
        C[k] ← 1
        FOR ogni adiacente j al nodo k DO          (Aggiorna i costi dei nodi adiacenti al nuovo nodo)
            IF C[j] = 0 AND COSTO[j] > peso dell'arco {k, j} THEN
                COSTO[j] ← peso dell'arco {k, j}
                T[j] ← k          (Aggiorna il padre)
            ENDIF
        ENDFOR
    ENDREPEAT
OUTPUT T.
```

L'inizializzazione richiede $O(|V|)$. Trovare un nodo di costo minimo fra quelli non ancora coperti richiede $O(|V|)$ (è sufficiente scorrere il vettore $COSTO$ tenendo conto del vettore C). Anche l'aggiornamento dei costi dei nodi adiacenti a quello scelto richiede al più $O(|V|)$. Ne segue che ogni iterazione del REPEAT richiede $O(|V|)$. Quindi, l'implementazione ha complessità asintotica $O(|V|^2)$.

Possiamo sperare di trovare una implementazione più efficiente? Per tentare di rispondere conviene, prima di tutto, cercare di capire quale potrebbe essere un limite invalicabile per la complessità asintotica delle implementazioni dell'algoritmo PRIM. Vale a dire, riuscire a mostrare che tutte le possibili implementazioni dell'algoritmo PRIM devono almeno avere una certa complessità asintotica. In generale, è estremamente difficile trovare buoni limiti inferiori. A tutt'oggi, eccettuati pochissimi esempi, il meglio che si riesce a fare è ottenere limiti inferiori sfruttando il semplice fatto che l'algoritmo deve perlomeno esaminare una certa parte dei dati di input. Che da questo tipo di analisi si ottenga un buon limite inferiore alla complessità dipende fortemente dall'algoritmo in questione. Facendo questo tipo di analisi dell'algoritmo PRIM ci si rende facilmente conto che l'algoritmo deve necessariamente esaminare tutti gli archi del grafo. Quindi, un limite inferiore alla complessità di possibili implementazioni dell'algoritmo PRIM è $O(|E|)$. A ben vedere questo è un limite inferiore alla complessità del problema MINIMO ALBERO DI COPERTURA, cioè, il limite si applica a qualsiasi algoritmo per detto problema. Il limite è equivalente a $O(|V| + |E|)$ dato che il grafo di input è supposto essere connesso. Confrontando il limite appena ottenuto con la complessità dell'implementazione sopra descritta si vede che l'implementazione è ottimale quando il grafo di input è molto denso, cioè, quando il numero di archi è $O(|V|^2)$. Quindi, non sarà possibile trovare una implementazione che è in tutti i casi più efficiente. Però potrebbe esistere una implementazione che è più efficiente quando il grafo non è denso, ad esempio quando il numero di archi è $O(|V|)$.

Ritorniamo quindi a riesaminare l'implementazione più in dettaglio per vedere dove è, eventualmente, possibile migliorarla. In ogni iterazione del REPEAT vengono svolte due operazioni: trovare un nodo di costo minimo ed aggiornare i costi dei nodi adiacenti al nodo trovato. I limiti sopra discussi alla complessità asintotica di queste due operazioni sono dei limiti *superiori*. Ovvero ogni esecuzione di tali operazioni richiede *al più* $O(|V|)$. Ma, siamo sicuri che tutte insieme le $|V|$ esecuzioni di queste operazioni richiedono non meno di $O(|V|^2)$ Per quanto riguarda l'operazione di trovare un nodo di costo minimo ciò è vero perché effettivamente il vettore $COSTO$ viene scandito per intero ad ogni esecuzione. Per l'operazione di aggiornamento la

complessità di una esecuzione dipende dal numero di nodi adiacenti a quello trovato. Più precisamente, se il nodo trovato è il nodo u allora l'esecuzione della relativa operazione di aggiornamento richiede $O(grado(u))$, dove $grado(u)$ è il grado di u, cioè, il numero di adiacenti al nodo u. Ne segue che tutte insieme le esecuzioni dell'operazione di aggiornamento richiedono qualcosa che è proporzionale alla somma dei gradi dei nodi del grafo. Ovvero richiedono $O(|E|)$. Quindi l'operazione di aggiornamento è implementata con complessità ottimale. Invece, l'operazione di trovare un nodo di costo minimo potrebbe forse essere migliorata. Una prima idea che viene in mente è di mantenere il vettore $COSTO$ ordinato, cosicché trovare il minimo richiederebbe $O(1)$. Però poi quando vengono effettuati gli aggiornamenti si devono spostare gli elementi del vettore e ciò può richiedere, e nel caso peggiore richiederà, $O(|V|)$. Così, miglioriamo l'operazione di trovare un nodo di costo minimo ma peggioriamo gli aggiornamenti e ritorniamo ad avere una complessità $O(|V|^2)$. Quello che ci occorre è una qualche struttura dati per mantenere i costi dei nodi che permetta di realizzare in modo efficiente due operazioni: trovare un costo minimo ed aggiornare i costi.

Un struttura semplice che forse fa al caso nostro è l'heap. Come si sa un heap permette di effettuare l'estrazione del minimo in $O(\log n)$, dove n è il numero di elementi dell'heap. Inoltre, anche l'aggiornamento di un valore richiede $O(\log n)$. Nel nostro caso i valori sono i costi dei nodi e l'aggiornamento consiste nel decremento di un costo. Occorre solamente fare attenzione a mantenere l'associazione tra i nodi e i relativi costi. Precisamente, occorre poter conoscere sia il nodo il cui costo è stato estratto dall'heap e sia la posizione nell'heap del costo di un nodo il cui costo deve essere aggiornato. Per questo si può usare un vettore P tale che $P[i]$ è la posizione all'interno dell'heap del costo del nodo i, se i è già stato estratto dall'heap allora $P[i] = 0$. Inoltre, l'heap può essere realizzato tramite un vettore H tale che $H[k].costo$ e $H[k].nodo$ sono rispettivamente il costo e il relativo nodo che si trova nella posizione k. Così per ogni nodo i presente nell'heap si ha che $H[P[i]].costo$ è il costo del nodo i e $H[P[i]].nodo = i$. La descrizione in pseudo-codice dell'implementazione tramite heap dell'algoritmo PRIM è la seguente:

```
INPUT Un vettore A[1...n] di liste di adiacenza con pesi di un grafo non diretto, connesso e con archi
    pesati
    T[1] ← 1                (Scegli come radice il nodo 1)
    H[1].costo ← 0           (Quindi, il nodo 1 ha costo 0 e)
    H[1].nodo ← 1            (si trova nella posizione 1 dell'heap)
    P[1] ← 1
    FOR i ← 2 TO n DO            (Tutti gli altri nodi hanno costo iniziale infinito)
        H[i].costo ← infinito
        H[i].nodo ← i
        P[i] ← i
    ENDFOR
    REPEAT n TIMES
        k ← H[1].nodo            (In k vi è il nodo di costo minimo)
        Aggiorna H e P a seguito dell'estrazione dall'heap del minimo.
        P[k] ← 0
        FOR ogni adiacente j a k DO
            IF P[j] ≠ 0 AND H[P[j]].costo > pesodell'arco{k, j} THEN
                H[P[j]].costo ← peso dell'arco {k, j}
                Aggiorna H e P a seguito del decremento del costo del nodo j
                T[j] ← k
            ENDIF
        ENDFOR
    ENDREPEAT
OUTPUT T.
```

L'inizializzazione richiede, come al solito, $O(|V|)$. Ad ogni iterazione del REPEAT l'estrazione del minimo richiede $O(\log |V|)$ e l'aggiornamento dei costi dei nodi adiacenti al nodo estratto u richiede al più $O(grado(u) \log |V|)$, dato che ogni decremento di costo richiede $O(\log |V|)$. Complessivamente questa implementazione richiede $O(|E| \log |V|)$. Come si vede le operazioni più onerose sono adesso gli aggiornamenti. Inoltre, non è sempre più efficiente dell'implementazione precedente. Quando il grafo non è molto denso l'implementazione tramite heap è più efficiente, ma quando il grafo è molto denso la semplice implementazione basata sul vettore dei costi risulta più efficiente. A questo punto ci si può chiedere se esiste una implementazione che è migliore di entrambe. La risposta è sì. Usando una versione raffinata di heap, chiamato heap di Fibonacci, è possibile implementare l'algoritmo PRIM con complessità $O(|E| + |V| \log |V|)$.

3.3. **Correttezza dell'Algoritmo KRUSKAL.** Passiamo ora ad analizzare l'algoritmo KRUSKAL. Anche per questo algoritmo dimostreremo la correttezza seguendo lo schema generale. Per comodità riportiamo l'algoritmo qui sotto:

INPUT Un grafo non diretto e connesso $G = (V, E)$ con archi pesati
  $SOL \leftarrow \emptyset$
  WHILE esiste un arco in E che può essere aggiunto a SOL senza creare cicli DO
    Sia $\{u, v\}$ un arco di peso minimo fra tutti quelli che possono essere aggiunti a SOL
    $SOL \leftarrow SOL \cup \{\{u, v\}\}$
  ENDWHILE
OUTPUT SOL.

Prima di tutto sinceriamoci che l'algoritmo termina sempre. Ad ogni iterazione del WHILE un nuovo arco viene aggiunto a SOL e la condizione del WHILE diventa falsa quando non ci sono più archi che possono essere aggiunti a SOL, quindi vengono eseguite al più $|E|$ iterazioni.

Sia m il numero di iterazioni eseguite. Sia SOL_h il valore di SOL al termine della h-esima iterazione e sia SOL_0 il valore iniziale. Dimostriamo che ogni soluzione parziale è estendibile ad una soluzione ottima.

Per ogni $h = 0, 1, \ldots, m$ esiste una soluzione ottima SOL^* che contiene SOL_h.

Dimostrazione. Procediamo per induzione su h. Per $h = 0$ è banalmente vero. Sia ora $h \geq 0$ (e $h < m$), assumendo la tesi vera per h la dimostriamo per $h + 1$. Quindi per ipotesi induttiva esiste una soluzione ottima SOL^* che contiene SOL_h. Se SOL^* contiene anche SOL_{h+1}, allora abbiamo fatto. Altrimenti, l'arco $\{u, v\}$ aggiunto a SOL_h nella $(h + 1)$-esima iterazione non appartiene a SOL^*. Cerchiamo di trasformare SOL^* in un'altra soluzione ottima che contiene SOL_{h+1}. Siccome gli archi di SOL^* formano un albero di copertura, in $SOL^* \cup \{u, v\}$ l'arco $\{u, v\}$ determina un ciclo C. L'arco $\{u, v\}$ non crea cicli con SOL_h per cui in C deve esserci almeno un'altro arco $\{x, y\}$ che non appartiene a SOL_h. Inoltre, l'arco $\{x, y\}$ non crea cicli con SOL_h dato che appartiene a SOL^* e SOL^* contiene SOL_h e non ha cicli. Quindi l'arco $\{x, y\}$ era tra gli archi che potevano essere scelti durante l'$(h + 1)$-esima iterazione. È stato scelto $\{u, v\}$ per cui deve essere che peso di $\{u, v\} \leq$ peso di $\{x, y\}$. Definiamo $SOL^\# = (SOL^* - \{\{x, y\}\}) \cup \{\{u, v\}\}$. Chiaramente, $SOL^\#$ è una soluzione ottima che contiene SOL_{h+1}.

Rimane da dimostrare che la soluzione finale SOL_m prodotta dall'algoritmo non solo è contenuta in una soluzione ottima SOL^* ma coincide con essa. Essendo m il numero di iterazioni eseguite dal WHILE deve essere che la condizione del WHILE risulta falsa per SOL_m. Quindi non ci sono archi nel grafo, e a maggior ragione in SOL^*, che possono essere aggiunti a SOL_m senza creare cicli. Siccome SOL_m è contenuto in SOL^* e SOL^* non ha cicli, allora non ci possono essere archi in SOL^* che non siano già in SOL_m. Dunque $SOL_m = SOL^*$.

3.4. **Implementazione dell'Algoritmo KRUSKAL.** Ora che sappiamo che l'algoritmo KRUSKAL è corretto passiamo ad esaminare possibili implementazioni efficienti. Osserviamo subito che, similmente all'algoritmo TERMINA_PRIMA, quando in una iterazione un arco è esaminato e viene scartato perché crea cicli non è possibile che venga successivamente aggiunto. Allora potrebbe risultare conveniente ordinare preliminarmente gli archi del grafo in ordine di peso non decrescente. Fatto ciò, per ogni arco che si presenta tramite questo ordine, si deve controllare se produce o meno cicli con gli archi in SOL. Per effettuare tale controllo osserviamo che SOL determina delle componenti connesse che coprono tutti i nodi del grafo e un arco non crea cicli con SOL se e solo se gli estremi dell'arco appartengono a componenti connesse differenti. Per mantenere l'informazione di queste componenti possiamo usare un vettore CC tale che, per ogni nodo i, $CC[i]$ è l'etichetta che identifica la componente a cui il nodo i appartiene. La descrizione in pseudo-codice di questa implementazione è la seguente:

INPUT Un vettore $A[1 \ldots n]$ di liste di adiacenza con pesi di un grafo non diretto, connesso e con archi pesati
  $SOL \leftarrow \emptyset$ (L'insieme di archi può essere implementato tramite una lista)
  Usa un vettore $E[1 \ldots m]$, dove m è il numero di archi, per ordinare gli archi in senso non decrescente di peso, tale che
  $E[h].u, E[h].v$ sono i nodi e $E[h].p$ è il peso dell'h-esimo arco e $E[1].p \leq E[2].p \leq \ldots \leq E[m].p$
  FOR $i \leftarrow 1$ TO n DO $CC[i] \leftarrow i$ (Inizialmente ogni nodo è una componente a se stante)
  FOR $h \leftarrow 1$ TO m DO
    IF $CC[E[h].u] \neq CC[E[h].v]$ THEN (Se gli estremi dell'arco appartengono a)
      $SOL \leftarrow SOL \cup \{\{E[h].u, E[h].v\}\}$ (componenti differenti, allora aggiungi l'arco)

```
            FOR i ← 1 TO n DO                    (e aggiorna il vettore delle componenti)
                IF CC[i] = CC[E[h].v] THEN CC[i] = CC[E[h].u]
            ENDFOR
        ENDIF
    ENDFOR
OUTPUT SOL.
```

L'ordinamento degli archi richiede $O(|E|\log|E|)$ tramite un algoritmo classico di ordinamento. Ogni iterazione del FOR sugli archi richiede un tempo che dipende fortemente dal fatto se l'arco crea o meno cicli. Se crea cicli l'iterazione richiede $O(1)$ ma se invece non crea cicli e quindi l'arco viene aggiunto l'iterazione richiede $O(|V|)$. Per fortuna però le iterazioni più onerose non sono così tante. Infatti, ce ne saranno tante quanti sono gli archi che vengono aggiunti a SOL, cioè, $|V|$–1. Quindi, il FOR sugli archi richiede $O(|E| + |V|^2) = O(|V|^2)$ e l'implementazione ha complessità asintotica $O(|E|\log|E| + |V|^2) = O(|E|\log|V| + |V|^2)$.

Ci si può chiedere se esiste una implementazione dell'algoritmo KRUSKAL con complessità $O(|E|\log|V|)$. Una tale implementazione esiste ed è basata su una struttura dati più raffinata per mantenere le componenti connesse che viene di solito chiamata Union-Find-Set.

4. Il Problema FILE ed Alcune Varianti Difficili

Si supponga di avere n file di lunghezze $l_1, l_2, \ldots, l_n$ (interi positivi) che bisogna memorizzare su un disco di capacità data D. La somma delle lunghezze dei file può eccedere la capacità del disco, così si vuole memorizzare il maggior numero possibile di file sul disco. L'idea dell'algoritmo greedy che si propone è molto semplice: ogni volta, scegli di memorizzare sul disco il file (tra quelli rimasti) con la minima lunghezza. Intuitivamente, questa è la scelta che lascia maggior spazio sul disco per gli altri file. Una descrizione in pseudo-codice dell'algoritmo è la seguente: l'algoritmo restituisce l'insieme degli indici dei file da memorizzare sul disco,

```
Algoritmo MIN_FILE
INPUT lunghezze di n file l_1, l_2, ..., l_n; capacità di un disco D
    SOL ← ∅
    F ← {1, 2, ..., n}
    M ← D
    WHILE F ≠ ∅ AND min{l_i | i ∈ F} ≤ M DO
        sia k l'indice di un file di lunghezza minima in F, cioè, l_k = min{l_i | i ∈ F}
        F ← F–{k}
        SOL ← SOL ∪ {k}
        M ← M–l_k
    ENDWHILE
OUTPUT SOL.
```

Dimostriamo la correttezza dell'algoritmo seguendo lo schema generale. Prima di tutto osserviamo che l'algoritmo termina, ossia il WHILE esegue un numero finito di iterazioni. Infatti, ad ogni iterazione viene tolto un indice da F, e quindi dopo al più n iterazioni la condizione del WHILE risulta falsa e il WHILE termina.

Sia m il numero di iterazioni del WHILE. Per ogni $h = 1, 2, \ldots, m$ sia SOL_h il valore di SOL al termine della h-esima iterazione del WHILE e sia $SOL_0 = \emptyset$ (il valore iniziale). Proviamo il seguente fatto.

Per ogni $h = 0, 1, 2, \ldots, m$ esiste una soluzione ottima SOL^* tale che SOL_h è incluso in SOL^*.

Dimostrazione. Innanzi tutto facciamo una semplice ma importante osservazione sui valori che via via sono assunti dalla variabile M. Per ogni $h = 1, 2, \ldots, m$ sia M_h il valore di M al termine della h-esima iterazione del WHILE e sia $M_0 = D$ (il valore iniziale). Siccome in ogni iterazione del WHILE ad M viene sotratta la lunghezza del file aggiunto a SOL ed inizialmente M è uguale alla capacità del disco, è chiaro che M_h è la capacità residua del disco dopo avervi memorizzato i file di SOL_h. Vale a dire $M_h = D - \sum_{k \in SOL_h} l_k$. Dimostriamo l'asserto per induzione su h. Per $h = 0$ l'enunciato è banalmente vero. Sia ora $h > 0$ e $h \leq m$. Assumendolo vero per h–1 vogliamo dimostrare che è vero anche per h. Sia SOL^* una soluzione ottima che contiene SOL_{h-1}. Se SOL^* contiene anche SOL_h allora abbiamo fatto. Altrimenti, deve essere che l'indice aggiunto a SOL nell'h-esima iterazione non è in SOL^*. Sia k tale indice. All'inizio dell'h-esima iterazione valeva $\min\{l_i | i \in F\} \leq M_{h-1}$ e k è stato scelto in modo tale che $l_k = \min\{l_i | i \in F\}$ perciò $l_k \leq M_{h-1}$ e quindi i file di SOL_h possono essere memorizzati sul disco. Siccome SOL^* è una soluzione ottima deve essere $|SOL^*| \geq |SOL_h|$. Inoltre, $SOL_h = SOL_{h-1} \cup \{k\}$ e SOL^* include SOL_{h-1}, ne segue che deve

esistere un indice j tale che j è in SOL^* e j non è in SOL_h. Siccome j non appartiene a SOL_h deve essere che j apparteneva ad F durante l'h-esima iterazione. L'indice k è stato scelto in modo da minimizzare la lunghezza tra quelli in F. Quindi $l_k \leq l_j$. Definiamo $SOL^{\#} = (SOL^* - \{j\}) \cup \{k\}$. Chiaramente $SOL^{\#}$ è ancora una soluzione ottima ed include SOL_h.

Adesso sappiamo che SOL_m, cioè la soluzione prodotta dall'algoritmo, è contenuta in una soluzione ottima SOL^*. Supponiamo per assurdo che SOL_m sia differente da SOL^*. Allora, esiste almeno un indice j tale che j è in SOL^* ma j non è in SOL_m. Siccome i soli indici che vengono tolti da F sono quelli che vengono inseriti in SOL, deve essere che j appartiene sempre ad F e in particolare quando il WHILE termina. Perciò quando la condizione del WHILE risulta falsa, F non è vuoto, quindi deve essere $\min\{l_i | i \in F\} > M$. Il valore di M, a quel punto dell'esecuzione dell'algoritmo, è uguale a $D - \sum_{k \in SOL_m} l_k$ e perciò si ha che

$$l_j \geq \min\{l_i | i \in F\} > M = D - \sum_{k \in SOL_m} l_k,$$

cioè, $l_j > D - \sum_{k \in SOL_m} l_k$. Inoltre, tenendo conto che $\sum_{i \in SOL^*} l_i \geq l_j + \sum_{k \in SOL_m} l_k$ si ha che

$$\sum_{i \in SOL^*} l_i \geq l_j + \sum_{k \in SOL_m} l_k > D - \sum_{k \in SOL_m} l_k + \sum_{k \in SOL_m} l_k$$

Quindi, $\sum_{i \in SOL^*} l_i > D$, ma ciò è in contraddizione con l'ipotesi che SOL^* sia una soluzione ammissibile (cioè, un insieme di file che può essere memorizzato sul disco).

La dimostrazione appena terminata sembra troppo laboriosa per provare qualcosa che in fin dei conti è "evidente". Infatti l'algoritmo, dopo tutto, non fa altro che scegliere i file di lunghezza più piccola, fino a quando non eccede la capacità del disco. Se l'algoritmo in tal modo sceglie un sottoinsieme, diciamo, di m file questi sono gli m file di lunghezza più piccola, cioè, se i file fossero ordinati per lunghezza non decrescente, l'algoritmo sceglierebbe i primi m file. Come è possibile che esista un'altro sottoinsieme di m file la cui somma delle lunghezze sia inferiore a quella del sottoinsieme scelto dall'algoritmo? La risposta che appare "evidente" è che non è possibile. Bene, allora si può basare una dimostrazione della correttezza dell'algoritmo su questa "evidenza". Vale a dire si può formalizzare, ovvero rendere rigorosa, la spiegazione del perchè tale "evidenza" è vera. Ma per ora supponiamo di aver già dimostrato la seguente proprietà:

dato un insieme di interi positivi $L = l_1, l_2, \ldots, l_n$ (le lunghezze dei file) la somma degli m interi più piccoli di L è minore od uguale alla somma di un qualsiasi sottoinsieme di L di cardinalità almeno m.

L'algoritmo MIN_FILE sceglie proprio i file di lunghezza più piccola finché la somma delle lunghezze non supera D. Quindi la soluzione SOL_m prodotta dall'algoritmo consiste degli m file di lunghezza più piccola ed m è tale che non è possibile aggiungere ad essa nessun'altro file senza superare la capacità del disco D. Supponiamo per assurdo che esista una soluzione ottima SOL^* che è migliore di SOL_m, cioè, $|SOL^*| > |SOL_m|$. Allora in SOL^* vi sono almeno $m+1$ file. Quindi c'e' almeno un file, diciamo il file j, che è in SOL^* ma non è in SOL_m. Siccome $|SOL^* - \{j\}| \geq m$, dalla proprietà suddetta deriva che

$$\sum_{k \in SOL_m} l_k \leq \sum_{k \in SOL^* - \{j\}} l_k.$$

Siccome SOL^* è una soluzione ottima e quindi ammissibile, deve essere che

$$\sum_{k \in SOL^* - \{j\}} l_k + l_j \leq D$$

.

Ma allora il file j si può aggiungere a SOL_m, una contraddizione.

La dimostrazione della proprietà suddetta è lasciata per esercizio. Guarda caso può essere fatta per induzione su m.

4.1. **Implementazione dell'Algoritmo MIN_FILE.** Una implementazione può essere data sulla falsa riga di quella per l'algoritmo TERMINA_PRIMA. Vale a dire, i file sono ordinati preliminarmente in ordine non decrescente di lunghezza e poi vengono scelti seguendo quest'ordine. Chiaramente, la complessità è $O(n \log n)$.

4.2. **Una Variante del Problema FILE e Problemi Computazionalmente Difficili.** Consideriamo la seguente variante del problema FILE che chiameremo FILE MEM. Dati n file di lunghezze $l_1, l_2, \ldots, l_n$ (interi positivi) e la capacità D di un disco si vuole trovare un sottoinsieme dei file che massimizza lo spazio di memoria usata sul disco. Più precisamente, si vuole trovare un sottoinsieme S dei file tale che

- $\sum_{i \in S} l_i \leq D$
- $\sum_{i \in S} l_i = \max\{\sum_{i \in F} l_i | F \subseteq \{1, 2, \ldots, n\} \text{ e } \sum_{i \in F} l_i \leq D\}$

Pensiamo ad un algoritmo greedy per FILE MEM. Chiaramente l'algoritmo MIN_FILE non è corretto. Infatti, basta considerare il caso di due soli file di lunghezze 1 e 5 e un disco di capacità 5. Quindi scegliere i file in ordine non descrescente di lunghezza non va bene. Forse scegliendo i file in ordine inverso, cioè dal più grande al più piccolo, avremo maggior fortuna. Però, riflettendoci un po' se i file vengono scelti seguendo un tale ordine potremmo scegliere un file grande che poi non permette di scegliere file più piccoli ma che saturano il disco. Infatti, potremmo avere tre file di lunghezze $8, 5, 4$ e un disco di capacità 9. L'algoritmo seglierebbe il primo file occupando 8 unità di memoria e lì si fermerebbe mentre chiaramente la soluzione ottima si ottiene scegliendo gli ultimi due file che occupano esattamente 9 unità di memoria. A questo punto risulta difficile immaginare altri algoritmi greedy per il problema FILE MEM. Sicuramente si potranno proporre altri algoritmi, magari molto più complicati o raffinati, ma altri algoritmi altrettanto semplici di quelli appena discussi non vengono in mente. Nonostante il problema FILE MEM abbia una formulazione così semplice e sia anche una semplice variante di un problema che abbiamo già risolto algoritmicamente con successo, esso è un problema difficile. Più precisamente, dovremmo dire computazionalmente difficile: non si conoscono algoritmi corretti per il problema FILE MEM che ammettano anche implementazioni efficienti. Naturalmente il problema, come quasi tutti i problemi di ottimizzazione, si può risolvere in modo esaustivo. Vale a dire tramite algoritmi che trovano una soluzione ottima facendo essenzialmente una ricerca esaustiva fra tutte le soluzioni possibili. Ma quante sono tutte le possibili soluzioni? Esse corrispondono a tutti i possibili sottoinsiemi dell'insieme dei file e quindi sono 2^n. Un numero che è quasi sempre esponenziale nella dimensione dell'istanza. Quindi un algoritmo esaustivo avrebbe una complessità esponenziale e quindi non può certo essere considerato efficiente. Abbiamo affermato che non si conoscono algoritmi efficienti per questo problema, ma siamo certi che non ne esistano? Invero no. A tutt'oggi nessuno è in grado di dimostrare che non esistono. Tuttavia, il problema FILE MEM appartiene ad una classe di problemi che è amplissima e studiatissima. I problemi di questa classe sono chiamati NP-completi. Le sorti computazionali dei problemi NP-completi sono strettamente legate fra loro. Se si riuscisse a trovare un algoritmo efficiente per uno qualsiasi di essi allora si troverebbero algoritmi efficienti per tutti i problemi NP-completi. E come conseguenza, se si riuscisse a dimostrare che per uno qualsiasi di essi non esistono algoritmi efficienti allora anche per gli altri problemi NP-completi non vi sarebbero speranze. La classe degli NP-completi comprende migliaia di problemi provenienti dalle aeree più disparate: informatica, matematica, logica, ingegneria, economia, teoria dei giochi, fisica, biologia, ecc. Per un gran numero di questi problemi sono stati cercati algoritmi efficienti per più di trentanni e si ricercano tutt'oggi. Tenendo conto che cercare un algoritmo efficiente per uno di questi problemi è come cercarlo per tutti i problemi della classe, la quantità complessiva di energie mentali spese per cercare algoritmi efficienti per uno o tutti i problemi della classe è enorme. Ciò costituisce la ragione empirica che fa propendere per l'ipotesi che invero algoritmi efficienti per tali problemi non esistano. Qui, a differenza di quanto discusso precedentemente, con il termine efficiente si intende una complessità che cresce al più come un polinomio nella dimensione dell'istanza. Questo significa che non solo non si conoscono algoritmi con complessità $O(n)$, $O(n^2)$, $O(n^3)$ ma nemmeno $O(n^{1000})$, dove n è la dimesione dell'istanza. Si conoscono soltanto algoritmi basati su ricerche di tipo esaustivo, magari anche raffinatissime, ma che nel caso peggiore hanno sempre una complessità che è esponenziale nella dimensione dell'istanza. Giusto per citare qualche altro esempio di problema appartenente alla classe che dovrebbe essere già noto: il problema della 3-colorazione di un grafo. Anche questo è una semplice variante di un problema molto più facile: il problema della 2-colorazione o della bipartizione di un grafo. Come è noto c'è un algoritmo semplice ed efficiente che risolve il problema di determinare se un grafo è bipartito o meno. Ciò equivale a chiedersi se i nodi del grafo possono essere colorati usando solo 2 colori in modo tale che nodi adiacenti abbiano colori differenti. Il problema della 3-colorazione consiste nel determinare, dato un grafo, se esiste una colorazione che usa al più 3 colori.

Qual'è la morale di tutto ciò? Ce ne sono diverse. Problemi apparentemente molto semplici e che sono magari anche molto vicini a problemi che si sanno risolvere algoritmicamente in modo efficiente, potrebbero non ammettere algoritmi efficienti, neanche lontanamente efficienti. Un'altra è che, quasi sempre, è estremamente difficile dimostrare che un problema non ammette un algoritmo efficiente.

4.3. **Una Variante del Problema SELEZIONE ATTIVITÀ.** Consideriamo ora una variante analoga a quella del problema FILE per il problema SELEZIONE ATTIVITÀ, che chiameremo UTILIZZAZIONE AULA. Date n attività (lezioni, seminari, ecc.) e per ogni attività i, $1 \leq i \leq n$, l'intervallo temporale $[s_i, f_i)$ in cui l'attività dovrebbe svolgersi, selezionare un insieme di attività che possono essere svolte senza sovrapposizioni in un'unica aula e che massimizzi il tempo totale di utilizzo dell'aula. Quindi in questo caso vogliamo massimizzare il tempo di utilizzo dell'aula da parte delle attività in essa

svolte, piuttosto che il numero di attività svolte. Iniziamo esaminando i tre algoritmi che avevamo proposto per il problema SELEZIONE ATTIVITÀ. L'algoritmo INIZIA_PRIMA sceglie le attività in ordine di tempo d'inizio. Un controesempio è presto trovato: un'attività che inizia prima di un'altra con essa incompatibile che però dura di più, ad esempio le due attività $[1, 3), [2, 5)$. Vediamo l'algoritmo DURA_MENO che sceglie le attività in ordine di durata crescente. Un controesempio è altrettanto facile di quello per l'algoritmo precedente: un'attività di breve durata incompatibile con un'altra attività che dura di più, ad esempio le due attività $[2, 4), [1, 5)$. Passiamo all'algoritmo TERMINA_PRIMA. Il controesempio che abbiamo dato per l'algoritmo INIZIA_PRIMA è un controesempio anche per l'algoritmo TERMINA_PRIMA. C'è almeno un'altro algoritmo che viene subito in mente: scegliamo le attività in ordine inverso rispetto all'algoritmo DURA_MENO, cioè, in ordine non crescente di durata. Se applicato ai controesempi dati sopra produce sempre la soluzione ottima. Più in generale è chiaro che se è applicato ad istanze con solo due attività l'algoritmo trova sempre la soluzione ottima. Se c'è un controesempio questo deve consistere di almeno tre attività. E in effetti basta pensare a una attività che è incompatibile con altre due attività fra loro compatibili ognuna di durata inferiore alla durata della prima ma la cui somma delle durate è invece maggiore di quella della prima. Ad esempio le tre attività $[3, 7), [2, 5), [5, 8)$. Si potrebbe continuare a proporre altri algoritmi, magari sempre più sofisticati o arzigogolati, ma crediamo che algoritmi greedy per il problema UTILIZZAZIONE AULA non esistano. Non abbiamo la certezza di ciò, è una congettura. Quello che si sa è che almeno un algoritmo efficiente esiste. Non è greedy ma usa una tecnica differente: la programmazione dinamica.

5. Il Problema CAMMINI MINIMI con Pesi Non Negativi

Supponiamo di voler trovare la strada più breve tra Roma e Trieste nella rete stradale italiana. Per fare ciò disponiamo di una cartina che per ogni tratto stradale compreso tra due incroci ne riporta la lunghezza in chilometri. Non è un compito così facile, potendo scegliere fra autostrade, strade statali, provinciali, ecc. e combinazioni qualsiasi di queste. Per chiarirci le idee, diamo una rappresentazione o modello della questione in termini matematici. La rete stradale può essere rappresentata con un grafo non diretto (assumiamo che tutte le strade siano percorribili in entrambi i sensi) e pesato. I nodi corrispondono agli incroci e gli archi ai tratti di strada compresi tra due incroci che sono privi al loro interno di incroci. Ogni arco ha un peso pari alla lunghezza del corrispondente tratto stradale. Se tra due incroci ci sono più tratti stradali senza incroci possiamo, senza perdere nulla, rappresentare nel grafo soltanto il tratto più breve. Così fra due nodi vi è al più un arco. Sia G_S il grafo così ottenuto. La questione a cui siamo interessati diventa: trovare in G_S un cammino di peso minimo fra tutti quelli che connettono i nodi r e t, dove r e t sono i nodi che rappresentano rispettivamente Roma e Trieste. Ricordiamo che in un grafo non diretto $G = (V, E)$ un cammino è una sequenza $u_1, u_2, ..., u_k$ di nodi tale che $\{u_1, u_2\}, \{u_2, u_3\}, \ldots, \{u_{k-1}, u_k\}$ sono archi in E, questi archi sono anche chiamati gli archi del cammino. In questo caso il cammino connette i nodi u_1 e u_k. Il peso di un cammino è semplicemente la somma dei pesi degli archi del cammino. Questa rappresentazione potrà essere usata anche per trovare la strada più breve fra altre coppie di città: Roma - Milano, Napoli - Bologna, ecc. Inoltre, il tipo di rappresentazione non cambia se invece di essere interessati a minimizzare la lunghezza della strada fossimo interessati a minimizzare il tempo di percorrenza o il costo del viaggio (carburante e/o pedaggi). Il tipo della rappresentazione, cioè un grafo non diretto e pesato in cui trovare cammini di peso minimo, rimane lo stesso anche se la rete stradale e quella di un'altra nazione o dell'intera europa o se invece della rete stradale siamo interessati ai collegamenti ferroviari o aerei o tramite altri mezzi. Inoltre, ci sono molte altre situazioni di cui tale tipo di rappresentazione fornisce un adeguato modello matematico.

La formulazione generale del problema è la seguente. Dato un grafo non diretto $G = (V, E)$ con archi pesati con pesi non negativi e due nodi s e t, trovare un cammino di peso minimo fra tutti quelli che connettono i nodi s e t. Il peso di un cammino di peso minimo che connette due nodi u e v può essere considerato come la distanza fra u e v, relativamente al grafo G. Formalmente possiamo definire:

$$dist_G(u, v) = \min\{P(C) | C \text{ è un cammino in } G \text{ che connette } u \text{ e } v\},$$

dove $P(C)$ denota il peso del cammino C. Se non ci sono cammini che connettono u e v conveniamo che $dist_G(u, v)$ sia infinito. Non a caso abbiamo usato il termine distanza, infatti $dist_G$ soddisfa le proprietà dell'essere una metrica (in effetti una pseudo-metrica): per tutti i nodi u, v e w,

(1) $dist_G(u, v) \geq 0$
(2) $dist_G(u, v) = dist_G(v, u)$
(3) $dist_G(u, v) \leq dist_G(u, w) + dist_G(w, v)$.

La proprietà che caratterizza meglio una distanza o metrica è la terza, la disuguaglianza triangolare, che formalizza l'idea intuitiva che la distanza che si percorre per andare da u a v passando anche per w è sempre maggiore od uguale della distanza (minima) da u a v. Vediamo perché $dist_G$ soddisfa la disuguaglianza triangolare. Se $dist_G(u, v)$ è infinito allora o $dist_G(u, w)$ o $dist_G(w, v)$ è infinito, altrimenti esisterebbe un cammino che connette u e v e quindi $dist_G(u, v)$ non sarebbe infinito. Se

$dist_G(u,v)$ non è infinito ed entrambi $dist_G(u,w)$ e $dist_G(w,v)$ non sono infinito, allora siano C_u e C_v dei cammini tali che $P(C_u) = dist_G(u,w)$ e $P(C_v) = dist_G(w,v)$. Concatenando C_u con C_v si ottiene un cammino C che connette u e v. Per la definizione di $dist_G$, risulta $dist_G(u,v) \leq P(C)$. Inoltre, $P(C) = P(C_u) + P(C_v) = dist_G(u,w) + dist_G(w,v)$, quindi $dist_G(u,v) \leq dist_G(u,w) + dist_G(w,v)$.

Torniamo al problema di trovare un cammino di peso minimo che connette s e t. Riflettiamo su algoritmi greedy per questo problema. Un algoritmo greedy dovrebbe costruire un cammino minimo da s a t gradualmente e senza mai cambiare una decisione già presa. Inoltre le decisioni che estendono passo per passo il cammino non possono basarsi su una elaborazione di tutte le possibili estensioni del cammino finora costruito. Ciò sarebbe computazionalmente troppo oneroso. Ma a questo punto ci si rende conto che tentare di costruire un singolo cammino, con queste limitazioni, porta inevitabilmente all'insuccesso. Infatti, se ad ogni passo estendiamo il cammino parziale con un arco, come scegliamo tale arco? Se non possiamo fare elaborazioni "in profondità" come possiamo essere sicuri di aver fatto la scelta giusta? Magari una scelta che localmente sembra la migliore può rivelarsi errata perché c'è un'altra scelta che, più avanti, apre una via molto più breve. Non possiamo limitarci a costruire un singolo cammino, dobbiamo necessariamente costruire più cammini contemporaneamente. Quindi partendo da s portiamo avanti più cammini fino a che uno di questi raggiunge il nodo finale t. Ma se non possiamo ritornare sulle decisioni già prese, allora i cammini parziali devono per forza essere dei cammini minimi. Ciò significa, ad un certo passo, si saranno calcolate le distanze da s di tutti i nodi raggiunti dai cammini parziali, fino a quel passo. Ricapitolando dovremmo trovare un modo efficiente per calcolare la distanza da s di un nuovo nodo conoscendo le distanze da s di alcuni altri nodi. All'inizio conosciamo solamente la distanza di s da s, cioè, $dist_G(s,s) = 0$. Di quale altro nodo possiamo conoscere facilmente la distanza da s? Il nodo che è adiacente ad s con peso minimo. Infatti, sia u un adiacente ad s con l'arco $\{s,u\}$ che ha peso minore od uguale al peso di un qualsiasi altro arco incidente in s. Allora $dist_G(s,u)$ =peso di $\{s,u\}$, dato che un qualsiasi cammino da s a u deve avere almeno un arco incidente in s. Ora che conosciamo anche la distanza di u da s, di quale altro nodo possiamo conoscere la distanza da s? Consideriamo un qualsiasi altro nodo v. Un cammino minimo C da s a v o passa per u o non passa per u. Se passa per u allora deve avere un arco incidente in u. Se invece non passa per u, deve avere un arco incidente in s. Quindi C contiene un nodo w che è adiacente o ad s o ad u. Siccome C è un cammino di peso minimo da s a v, il cammino C fino a w deve essere un cammino di peso minimo da s a w. Consideriamo allora tutti i nodi che sono adiacenti o ad s o ad u. Fra questi c'è almeno un nodo per cui un cammino minimo da s consiste o di un arco incidente in s o di un cammino minimo fino ad u concatenato con un arco incidente in u. Sì ma quale? Beh è semplice, quello che fra tutti gli adiacenti o ad s o ad u avrebbe la distanza da s minima se così calcolata. Vale a dire scegliamo un nodo v per cui esiste un arco $\{x,v\}$ con x uguale o a s o a u e tale che

$$dist_G(s,x) + p\{x,v\} = \ min\{dist_G(s,y) + p\{y,w\} |\ y \text{ è o } s \text{ o } u \text{ e } \{y,w\} \text{ è un arco di } G\},$$

dove $p\{a,b\}$ denota il peso dell'arco $\{a,b\}$. Dovrebbe essere chiaro come generalizzare questo criterio. Ma prima di descrivere l'algoritmo e dimostrarne la correttezza, diamo la definizione del problema nella forma che più si aggrada all'algoritmo. In effetti l'algoritmo tratteggiato non solo calcolerebbe la distanza fra s e t ma anche le distanze fra s e gran parte dei nodi del grafo. Anzi, siccome le distanze sarebbero calcolate in ordine non decrescente, quando la distanza fra s e t è molto grande verrebbero calcolate tutte o quasi le distanze fra s e gli atri nodi. Questo suggerisce di considerare il seguente problema, chiamato CAMMINI MINIMI (con pesi non negativi). Dato un grafo non diretto $G = (V,E)$ con archi pesati con pesi non negativi e un nodo s, trovare per ogni nodo u raggiungibile da s un cammino di peso minimo fra quelli che connettono s e u. Per ora consideremo solamente il calcolo delle distanze e successivamente vedremo come modificare l'algoritmo per ottenere anche i cammini minimi. Ecco la descrizione in pseudo codice dell'algoritmo sopra tratteggiato:

INPUT un grafo non diretto $G = (V,E)$ con archi pesati con pesi non negativi e un nodo s
  $d[s] \leftarrow 0$ (La distanza di s da s è 0)
  FOR ogni nodo u diverso da s DO $d[u] \leftarrow$ infinito
  $R \leftarrow \{s\}$ (Mantiene l'insieme dei nodi di cui si è già calcolata la distanza da s)
  WHILE esiste un arco con un estremo in R e l'altro fuori di R DO
    Sia $\{u,v\}$ un arco in E tale che $u \in R$, $v \notin R$ e $d[u]+p\{u,v\} = \min\{d[x]+p\{x,y\} | x \in R,\ y \notin R \text{ e } \{x,y\} \in E\}$
    $d[v] \leftarrow d[u] + p\{u,v\}$
    $R \leftarrow R \cup \{v\}$
  ENDWHILE
OUTPUT d.

Dimostriamo che l'algoritmo è corretto, cioè, calcola le distanze in G dal nodo s. Per prima cosa sinceriamoci che l'algoritmo termina sempre. In effetti, se la condizione del WHILE è vera allora l'insieme in cui è effettuata la scelta di un

arco non è vuoto e quindi ad ogni iterazione è aggiunto un nuovo nodo ad R. Ne segue che il WHILE non può eseguire più di $|V|$–1 iterazioni.

Procediamo ora seguendo lo schema generale. In questo caso una soluzione parziale è rappresentata da un vettore parziale delle distanze, cioè, un vettore definito solamente in alcuni elementi. L'insieme in cui è definito è dato proprio da R. Così, denotiamo con d_h ed R_h rispettivamente il vettore d e l'insieme R al termine della h-esima iterazione. Come al solito d_0 e R_0 denotano i valori iniziali. Sia k il numero di iterazioni eseguite. Dobbiamo dimostrare che d_h è estendibile ad una soluzione ottima. In questo caso c'è un'unica soluzione ottima che è $dist_G$.

Per ogni $h = 0, 1, \ldots, k$, d_h è uguale a $dist_G$ su R_h, cioè, per ogni u in R_h, $d_h[u] = dist_G[u]$.

Dimostrazione. Procediamo per induzione su h. Per $h = 0$, è vero perché $R_0 = \{s\}$ e $d_0[s] = 0 = dist_G[s]$. Sia ora $0 \leq h < k$. Per ipotesi induttiva, per ogni u in R_h, $d_h[u] = dist_G[u]$. Vogliamo allora far vedere che la tesi è vera anche per $h + 1$. Sia $\{u, v\}$ l'arco che è stato scelto nella $(h + 1)$-esima iterazione. Per cui $R_{h+1} = R_h \cup \{v\}$. Dobbiamo quindi dimostrare che $d_{h+1}[v] = dist_G[v]$. Per l'ipotesi induttiva $d_h[u] = dist_G[u]$ per cui esiste un cammino C_u da s a u di peso $d_h[u]$. Sia C_v il cammino da s a v che si ottiene concatenando C_u con l'arco $\{u, v\}$. Siccome $d_{h+1}[v] = d_h[u] + p\{u, v\}$, il peso di C_v è proprio pari a $d_{h+1}[v]$. Quindi deve essere $d_{h+1}[v] \geq dist_G[v]$. Rimane da far vedere la disuguaglianza opposta. Sia C^* un cammino di peso minimo da s a v, cioè, $P(C^*) = dist_G[v]$. Percorrendo il cammino C^* da s verso v, sia z il primo nodo che s'incontra che non appartiene a R_h. Siamo sicuri che z esiste perché almeno l'ultimo nodo di C^*, cioè v, non appartiene a R_h. Sia w il predecessore di z in C^*. Siamo sicuri che w esiste perchè C^* inizia in s e siccome s appartiene a R_h non può essere che z sia proprio s. Tutto ciò implica che w è in R_h, z non è in R_h ed esiste l'arco $\{w, z\}$. Quindi l'arco $\{w, z\}$ era uno degli archi che potevano essere scelti nella $(h + 1)$-esima iterazione. Siccome è stato scelto $\{u, v\}$, deve essere $d_h[u] + p\{u, v\} \leq d_h[w] + p\{w, z\}$. Sia C_w la parte del cammino C^* che va da s a w. Chiaramente $P(C_w) \geq dist_G[w]$. Per l'ipotesi induttiva $d_h[w] = dist_G[w]$. Inoltre, $P(C^*) \geq P(C_w) + p\{w, z\}$, dato che il peso degli eventuali altri archi di C^* è non negativo. Mettendo insieme quanto finora dimostrato si ottiene:

$$dist_G[v] = P(C^*) \geq P(C_w) + p\{w, z\} \geq d_h[w] + p\{w, z\} \geq d_h[u] + p\{u, v\} = d_{h+1}[v],$$

e la dimostrazione è completa.

Ora sappiamo che, per ogni u in R_k, $d_k[u] = dist_G[u]$, e d_k è proprio il vettore delle distanze calcolate dall'algoritmo. Rimane da far vedere che se v non è in R_k allora $dist_G[v]$ è infinito. Siccome il WHILE ha eseguito esattamente k iterazioni, questo vuol dire che la condizione del WHILE è falsa per R_k. Ciò significa che non esistono archi di G con un estremo in R_k e l'altro fuori da R_k. Quindi se un nodo v non appartiene a R_k non ci possono essere cammini che connettono s e v, dato che s è in R_k. Dunque $dist_G[v]$ è infinito.

Adesso sappiamo che l'algoritmo calcola le distanze corrette. Vorremmo che l'algoritmo trovasse anche i cammini di peso minimo. Ma questo non è difficile. Infatti, l'algoritmo implicitamente già costruisce dei cammini minimi. Ogni volta che sceglie un arco quello è un arco che appartiene a uno o più cammini minimi. Più precisamente quando sceglie un arco $\{u, v\}$ allora questo è l'ultimo arco di un cammino minimo per il nodo v. Tale cammino è la concatenazione del cammino minimo, precedentemente trovato, per il nodo u con l'arco $\{u, v\}$. A questo punto dovrebbe essere intuitivamente evidente, ma lo rivedremo più in dettaglio dopo, che i cammini così costruiti formano un albero radicato in s. Quindi per rappresentare tali cammini possiamo usare un vettore dei padri. La descrizione dell'algoritmo modificato, noto come algoritmo di Dijkstra, è la seguente:

Algoritmo DIJKSTRA
INPUT un grafo non diretto $G = (V, E)$ con archi pesati con pesi non negativi e un nodo s
    $d[s] \leftarrow 0$ (La distanza di s da s è 0)
    $T[s] \leftarrow s$
    FOR ogni nodo u diverso da s DO $d[u] \leftarrow$ infinito
    $R \leftarrow \{s\}$ (Mantiene l'insieme dei nodi di cui si è già calcolata la distanza da s)
    WHILE esiste un arco con un estremo in R e l'altro fuori di R DO
        Sia $\{u, v\} \in E$ tale che $u \in R$, $v \notin R$ e $d[u] + p\{u, v\} = \min\{d[x] + p\{x, y\} | x \in R,\ y \notin R$ e $\{x, y\} \in E\}$
        $d[v] \leftarrow d[u] + p\{u, v\}$
        $T[v] \leftarrow u$
        $R \leftarrow R \cup \{v\}$

ENDWHILE
OUTPUT d, T.

Dobbiamo dimostrare che effettivamente T rappresenta dei cammini minimi da s a tutti i nodi raggiungibili da s. Sia ancora R_k l'insieme dei nodi raggiungibili da s in G. Avendo già dimostrato che d_k dà le distanze corrette, sarà sufficiente provare che T rappresenta un albero radicato in s i cui nodi sono esattamente i nodi in R_k, i cui archi sono archi di G e tale che per ogni nodo u in R_k il peso del cammino da s a u in T è uguale a $d_k[u]$. Ciò può essere dimostrato per induzione sulle iterazioni del WHILE. Sia T_h il vettore T al termine della h-esima iterazione ristretto ai nodi in R_h e sia T_0 il vettore iniziale ristretto a R_0.

Per ogni $h = 0, 1, \ldots, k$, T_h rappresenta un albero radicato in s i cui nodi sono esattamente i nodi in R_h, i cui archi sono archi di G e tale che per ogni nodo u in R_h il peso del cammino da s a u in T_h è uguale a $d_h[u]$.

Dimostrazione. Procediamo per induzione su h. Per $h = 0$, T_0 rappresenta un albero costituito dal solo nodo s, R_0 è proprio uguale a $\{s\}$ e chiaramente il cammino da s a s in T_0 ha peso 0 che è uguale a $d_0[s]$. Sia ora $0 \leq h < k$, assumendo la tesi vera per h la vogliamo dimostrare per $h + 1$. Sia $\{u, v\}$ l'arco scelto nella $(h + 1)$-esima iterazione. Il vettore T_{h+1} è uguale a T_h eccetto che è definito anche per il nuovo nodo v che è aggiunto ad R_h e per ipotesi induttiva T_h rappresenta un albero radicato in s i cui nodi sono esattamente i nodi in R_h. Da ciò e dal fatto che u è in R_h, v non è in R_h e $T_{h+1}[v] = u$, deriva che T_{h+1} rappresenta un albero radicato in s i cui nodi sono esattamente i nodi in R_{h+1}. Gli archi di T_{h+1} sono archi di G perchè per ipotesi induttiva gli archi di T_h sono archi di G e l'unico arco in più in T_{h+1} rispetto a T_h è l'arco $\{u, v\}$ che ovviamente appartiene a G. Sempre per ipotesi induttiva si ha che per ogni nodo u in R_h il peso del cammino da s a u in T_h è uguale a $d_h[u]$. Siccome T_{h+1} differisce da T_h soltanto sul nuovo nodo v, vale che per ogni nodo u in R_h il peso del cammino da s a u in T_{h+1} è uguale a $d_h[u]$, che coincide con $d_{h+1}[u]$. Inoltre il cammino da s a v in T_{h+1} è dato dal cammino da s a u in T_{h+1} concatenato con l'arco $\{u, v\}$. Quindi il peso del cammino è $d_h[u] + p\{u, v\} = d_{h+1}[v]$ e questo completa la dimostrazione.

Una delle conseguenze dell'algoritmo DIJKSTRA, che non è immediatamente evidente, è che i cammini minimi da un nodo s a tutti gli altri nodi raggiungibili da s possono sempre essere scelti in modo tale che la loro unione formi un albero radicato in s.

Per ricostruire i cammini minimi dal vettore dei padri T c'è una procedura ricorsiva molto semplice:

Procedura CAMMINO
INPUT vettore dei padri T, nodo sorgente s e un nodo destinazione u
IF $u \neq s$ THEN CAMMINO$(T, s, T[u])$
Stampa u
END.

Quindi è possibile ricostruire il cammino in $O(|V|)$.

5.1. **Implementazione dell'Algoritmo DIJKSTRA.** Dovrebbe essere abbastanza evidente che l'algoritmo DIJKSTRA ha una forma molto simile all'algoritmo PRIM. Non è quindi sorprendente che si possono dare per l'algoritmo DIJKSTRA implementazioni che sono della stessa forma di quelle che abbiamo già visto per l'algoritmo PRIM. Infatti l'unica cosa che cambia è come è definito il costo di un nodo. Nel caso dell'algoritmo PRIM il costo di un nodo v è il minimo fra i pesi degli archi che connettono v ai nodi già scelti. Mentre nell'algoritmo DIJKSTRA il costo di un nodo v è il minimo fra le "distanze possibili" relativamente ai nodi già scelti. Dove per "distanza possibile" di un nodo v intendiamo un qualsiasi valore $d[u]+p\{u, v\}$ tale che u è un nodo già scelto ed $\{u, v\}$ è un arco del grafo. Quindi sostituendo questo costo per il costo dei nodi usato nelle implementazioni dell'algoritmo PRIM si ottengono delle implementazioni per l'algoritmo DIJKSTRA con complessità $O(|V|^2)$, $O(|E| \log |V|)$ e $O(|V| \log |V| + |E|)$

6. Il Problema ZAINO e Algoritmi di Approssimazione

Consideriamo il seguente problema chiamato ZAINO. Dato uno zaino di capacità C ed n oggetti con valori $v_1, v_2, \ldots, v_n$ e pesi $p_1, p_2, \ldots, p_n$, selezionare un sottoinsieme degli n oggetti con la proprietà che il peso totale degli oggetti selezionati non superi la capacità dello zaino e il valore totale sia massimo. Più precisamente, si vuole trovare un sottoinsieme Z degli oggetti tale che

- $\sum_{i \in Z} p_i \leq C$

- $\sum_{i\in Z} v_i = \max\{\sum_{i\in S} v_i |\ S$ sottoinsieme di $\{1,2,\ldots,n\}$ e $\sum_{i\in S} p_i \leq C\}$.

Di solito per problemi di tal fatta si preferisce rappresentare un sottoinsieme tramite il corrispondente vettore caratteristico. Con questo tipo di rappresentazione ciò che il problema richiede può essere formulato nel seguente modo: trovare un vettore $x_1, x_2, ..., x_n$, con x_i pari a 0 o 1 tale che

- $\sum_{i=1}^{n} x_i p_i \leq C$
- $\sum_{i=1}^{n} x_i v_i = \max\{\sum_{i=1}^{n} y_i v_i |\ y_i$ è 0 o 1 e $\sum_{i=1}^{n} y_i p_i \leq C\}$.

Esaminiamo qualche idea per un algoritmo greedy. Si tratta di trovare un modo opportuno per scegliere gli oggetti. Una possibilità è quella di preferire fra tutti gli oggetti non ancora considerati quello di massimo valore, sempreché non si ecceda la capacità dello zaino. Però potrebbero esserci più oggetti di minor valore che a parità di peso valgono di più dell'oggetto di valore massimo. Ad esempio consideriamo l'istanza:

	1	2	3	
v	7	6	5	
p	4	3	2	$C = 5$

L'algoritmo proposto sceglie l'oggetto 1 con valore 7 e poi non può più prendere altri oggetti. Ma la soluzione ottima consiste nello scegliere gli oggetti 2 e 3 con valore totale 11.

Preferire fra tutti gli oggetti non ancora considerati quello di peso minimo è un approccio destinato anch'esso a fallire in quanto può portare a preferire oggetti di scarso valore sebbene di modico peso. Infatti consideriamo l'istanza:

	1	2	3	
v	6	10	22	
p	1	2	3	$C = 5$

Chiaramente l'algoritmo proposto sceglie gli oggetti 1 e 2 con valore totale 16, mentre la soluzione ottima consiste nello scegliere gli oggetti 1 e 3 con valore totale 28.

Dovrebbe essere chiaro a questo punto che nella scelta del nuovo oggetto bisogna tener conto di entrambi i parametri che lo caratterizzano. Una volta considerato ciò è naturale pensare di scegliere gli oggetti in base al valore per unità di peso. Vale a dire scegliamo fra tutti gli oggetti rimasti quello che ha il massimo rapporto valore/peso e che ovviamente non ecceda la capacità dello zaino. Diamo subito una descrizione di questo algoritmo che sembra essere molto promettente:

Algoritmo ZAINO
INPUT valori $v_1, v_2, ..., v_n$ e pesi $p_1, p_2, ..., p_n$ di n oggetti e la capacità C di uno zaino
$peso \leftarrow 0$
$R \leftarrow \{1, 2, \ldots, n\}$
WHILE $R \neq \emptyset$ DO
  sia k l'oggetto in R con massimo rapporto valore/peso, cioè, $\frac{v_k}{p_k} = \max\{\frac{v_i}{p_i} | i \in R\}$
  IF $peso + p_k \leq C$ THEN $SOL[k] \leftarrow 1$
  ELSE $SOL[k] \leftarrow 0$
  $peso \leftarrow peso + SOL[k] \cdot p_k$
  $R \leftarrow R$–$\{k\}$
ENDWHILE
OUTPUT SOL.

Nonostante il criterio di scelta adottato dall'algoritmo sembri il migliore possibile, un po' di riflessione ci porta ad intuire che potrebbero esserci dei casi che mettono in difficoltà l'algoritmo. In effetti, potrebbero esserci degli oggetti con un alto rapporto valore/peso che una volta scelti riempano poco lo zaino e però non permettono che si possa aggiungere altri oggetti. Mentre potrebbero esserci altri oggetti con un rapporto valore/peso leggermente inferiore ma che riempono meglio lo zaino. Ad esempio, consideriamo la seguente istanza:

	1	2	3	
v	6	10	12	
p	1	2	3	$C = 5$

L'algoritmo ZAINO sceglie gli oggetti 1 e 2 con valore totale 16, mentre l'ottimo è dato dagli oggetti 2 e 3 con valore totale 22.

A ben vedere sarebbe stato sorprendente se l'algoritmo ZAINO fosse stato corretto per il problema omonimo. Infatti, la variante del problema FILE, di cui abbiamo discusso la difficoltà precedentemente, può essere vista come un caso particolare del problema ZAINO. È sufficiente che sia i valori che i pesi degli oggetti siano uguali alle lunghezze dei file. In questo caso i rapporti valore/peso sono tutti uguali a 1. Quindi il criterio che guida le scelte dell'algoritmo diventa del tutto ininfluente.

L'algoritmo ZAINO non è corretto però potrebbe produrre comunque soluzioni che sebbene non ottime potrebbero essere vicine a soluzioni ottime. Ad esempio, la soluzione prodotta dall'algoritmo potrebbe avere sempre un valore totale che è almeno la metà di quello di una soluzione ottima. O magari potrebbe essere ancora più vicina all'ottimo. Prima però di affrontare la questione conviene premettere una breve discussione su i cosidetti algoritmi di approssimazione e più in generale su algoritmi che sebbene non corretti possono comunque risultare utili nella risoluzione di problemi computazionalmente difficili.

6.1. **Problemi Difficili ed Algoritmi di Approssimazione.** Si è già accennato precedentemente all'esistenza di moltissimi problemi, che hanno grande importanza sia sul piano teorico sia su quello applicativo, ma che sono computazionalmente difficili. Ovvero non si conoscono algoritmi neanche lontanamente efficienti per tali problemi. Tuttavia, spesso trovare la soluzione ottima non è l'unica cosa che interessa. Potrebbe essere già soddisfacente ottenere una soluzione che sia soltanto vicina ad una soluzione ottima e, ovviamente, più è vicina e meglio è. In realtà è proprio così per un gran numero di problemi.

Fra gli algoritmi che non trovano sempre una soluzione ottima, è importante distinguere due categorie piuttosto differenti. Ci sono gli algoritmi per cui si *dimostra* che la soluzione prodotta ha almeno una certa vicinanza ad una soluzione ottima. In altre parole, è garantito che la soluzione prodotta approssima entro un certo grado una soluzione ottima. Questi sono chiamati *algoritmi di approssimazione* e vedremo tra poco come si misura la vicinanza ad una soluzione ottima. L'altra categoria è costituita da algoritmi per cui non si riesce a dimostrare che la soluzione prodotta ha sempre una certa vicinanza ad una soluzione ottima. Però, sperimentalmente sembrano comportarsi bene. Essi sono a volte chiamati *algoritmi euristici*. Spesso sono l'ultima spiaggia, quando non si riesce a trovare algoritmi corretti efficienti né algoritmi di approssimazione efficienti che garantiscano un buon grado di approssimazione, rimangono soltanto gli algoritmi euristici. In realtà, per una gran parte dei problemi NP-completi non solo non si conoscono algoritmi corretti efficienti ma neanche buoni algoritmi di approssimazione. Non è quindi sorprendente che fra tutti i tipi di algoritmi, gli algoritmi euristici costituiscano la classe più ampia e che ha dato luogo ad una letteratura sterminata. Ciò è dovuto non solo ai motivi sopra ricordati ma anche, e forse soprattutto, al fatto che è quasi sempre molto più facile inventare un nuovo algoritmo o una variante di uno già esistente e vedere come si comporta sperimentalmente piuttosto che dimostrare che un algoritmo, vecchio o nuovo che sia, ha una certa proprietà (ad esempio, è corretto, garantisce un certo grado di approssimazione, ecc.). Prima di passare a descrivere più in dettaglio gli algoritmi di approssimazione, è bene osservare che la classificazione che abbiamo dato in algoritmi di approssimazione ed algoritmi euristici è rispetto alla realtà una semplificazione. Infatti esistono algoritmi che non ricadono in nessuna delle classi discusse. Ad esempio gli *algoritmi probabilistici* per cui si dimostra che con alta probabilità producono una soluzione ottima ma che possono anche produrre soluzioni non ottime sebbene con piccola probabilità. Un'altro importante esempio sono gli algoritmi che producono sempre una soluzione ottima ma non è garantito che il tempo di esecuzione sia sempre limitato da un polinomio.

Un algoritmo di approssimazione per un dato problema è un algoritmo per cui si dimostra che la soluzione prodotta approssima sempre entro un certo grado una soluzione ottima per il problema. Si tratta quindi di specificare cosa si intende per "approssimazione entro un certo grado". Ci occorrono alcune semplici nozioni. Sia Π un qualsiasi problema di ottimizzazione e sia X una istanza di Π indichiamo con $OTT_\Pi(X)$ il valore o misura di una soluzione ottima per l'istanza X di Π. Ad esempio se Π è il problema SELEZIONE ATTIVITÀ ed X è un istanza di tale problema allora $OTT_\Pi(X)$ è il numero massimo di attività di X che possono essere svolte senza sovrapposizioni in un unica aula. Se Π è il problema MINIMO ALBERO DI COPERTURA ed X è un istanza di tale problema allora $OTT_\Pi(X)$ è il peso di un albero di copertura di peso minimo per X. Quindi $OTT_\Pi(X)$ non denota una soluzione ottima per l'istanza X ma soltanto il valore di una soluzione ottima. Chiaramente tutte le soluzioni ottime per una certa istanza X hanno lo stesso valore. Sia A un algoritmo per un problema Π e sia X un instanza di Π, indichiamo con $A_\Pi(X)$ il valore o misura della soluzione prodotta dall'algoritmo A con input l'istanza X. Il pedice Π in $A_\Pi(X)$ serve a ricordare che il valore della soluzione prodotta da A dipende dal problema Π. Il modo usuale di misurare il grado di aprossimazione di un algoritmo non è altro che il rapporto fra il valore di una soluzione ottima e il valore della soluzione prodotta dall'algoritmo. Assumeremo che un algoritmo produca perlomeno una soluzione che è ammissibile. Per fare sì che il rapporto dei valori dia sempre un numero maggiore od uguale a 1, si distingue fra problemi di minimo e problemi di massimo. Ovvero se una soluzione per il problema è ottima perchè massimizza una certa quantità (come ad esempio SELEZIONE ATTIVITÀ) oppure la minimizza (come ad esempio MINIMO ALBERO DI

COPERTURA). Iniziamo dai problemi di massimo. Sia Π un problema di massimo ed A un algoritmo per Π. In questo caso risulta sempre $A_\Pi(X) \leq OTT_\Pi(X)$. Si dice che A approssima Π entro un fattore di approssimazione r se

$$\text{per ogni istanza } X \text{ di } \Pi, \ \frac{OTT_\Pi(X)}{A_\Pi(X)} \leq r.$$

Analogamente per i problemi di minimo. Sia Π un problema di minimo ed A un algoritmo per Π. In questo caso risulta sempre $A_\Pi(X) \geq OTT_\Pi(X)$. Si dice che A approssima Π entro un fattore di approssimazione r se

$$\text{per ogni istanza } X \text{ di } \Pi, \ \frac{A_\Pi(X)}{OTT_\Pi(X)} \leq r.$$

Quindi se A approssima Π entro un fattore 1 ciò equivale a dire che A è corretto per Π, cioè trova sempre una soluzione ottima. Se invece A approssima Π entro, ad esempio, un fattore 22, ciò significa che A trova sempre una soluzione di valore almeno pari alla metà di quello di una soluzione ottima, se Π è un problema di massimo, e al più pari al doppio di quello di una soluzione ottima se Π è di minimo.

6.2. **Algoritmi di Approssimazione per il Problema ZAINO..** Entro quale fattore l'algoritmo ZAINO approssima il problema omonimo? Consideriamo la seguente istanza:

	1	2	
v	1	99	
p	1	100	$C = 100$

L'algoritmo ZAINO produce la soluzione costituita dall'oggetto 1 di valore 1. Mentre la soluzione ottima è data dall'oggetto 2 di valore 99. Per questa istanza l'algoritmo sbaglia di un fattore 99. Chiaramente questo esempio può essere generalizzato e per ogni fissato fattore f esiste una istanza per cui l'algoritmo sbaglia di un fattore almeno pari ad f. Questo significa che per l'algoritmo ZAINO non esiste alcuna costante, per quanto grande, per cui si possa dire che l'algoritmo approssima il problema ZAINO entro tale costante. L'algoritmo ZAINO non approssima il problema ZAINO neanche entro un fattore pari ad 1000000. E quel che è peggio e che ci sono istanze molto piccole per cui sbaglia tantissimo. Però conosciamo un caso particolare del problema ZAINO che invece ammette un algoritmo corretto. Il problema FILE può essere riformulato come un sottoproblema del problema ZAINO. Infatti, basta considerare i file come oggetti con valori tutti pari ad 1 e con pesi uguali alle relative lunghezze. Inoltre, è facile verificare che l'algoritmo ZAINO coincide con l'algoritmo MIN_FILE su queste particolari istanze. Quindi sappiamo che quando l'istanza ha i valori degli oggetti tutti uguali, allora l'algoritmo ZAINO produce sempre una soluzione ottima. In effetti, le istanze per cui abbiamo visto che l'algoritmo invece sbaglia tantissimo hanno, in un certo senso, la caratteristica opposta: ci sono oggetti che hanno valori molto diversi fra loro. Questo fa pensare che su istanze in cui i valori degli oggetti variano di poco l'algoritmo ZAINO potrebbe garantire un fattore di approssimazione. Allo scopo di esaminare più a fondo il comportamento dell'algoritmo, analizziamone il comportamento relativamente ad una variante "rilassata" del problema ZAINO.
Consideriamo la seguente variante del problema ZAINO che chiameremo ZAINO FRAZIONARIO. Data una capacità C ed n oggetti con valori $v_1, v_2, \ldots, v_n$ e pesi $p_1, p_2, \ldots, p_n$, trovare un vettore $x_1, x_2, \ldots, x_n$, con x_i numero reale compreso fra 0 e 1 tale che

- $\sum_{i=1}^{n} x_i p_i \leq C$
- $\sum_{i=1}^{n} x_i v_i = \max\{\sum_{i=1}^{n} y_i v_i |\ 0 \leq y_i \leq 1 \text{ e } \sum_{i=1}^{n} y_i p_i \leq C\}$.

Quindi l'unica differenza con il problema originale è che si possono scegliere anche frazioni di oggetti. Ad esempio consideriamo la seguente istanza del nuovo problema:

	1	2	3	
v	6	10	12	
p	1	2	3	$C = 5$

Allora la versione "frazionaria" dell'algoritmo ZAINO sceglie gli oggetti 1 e 2 per intero e i 2/3 dell'oggetto 3 totalizzando $6 + 10 + 2/3 \cdot 12 = 24$. Non a caso abbiamo anche parlato di variante "rilassata". Il termine "rilassata" si riferisce al fatto che i vincoli sul vettore $x_1, x_2, \ldots, x_n$ sono stati appunto rilassati, cioè, originariamente ogni x_i doveva essere un intero fra 0 e 1 mentre nella variante può assumere un qualsiasi valore reale compreso fra 0 e 1. La versione dell'algoritmo ZAINO per questa variante è immediata:

Algoritmo ZAINO_FRAZ
INPUT valori $v_1, v_2, \ldots, v_n$ e pesi $p_1, p_2, \ldots, p_n$ di n oggetti e una capacità C
$peso \leftarrow 0$
$R \leftarrow \{1, 2, \ldots, n\}$

WHILE $R \neq \emptyset$ DO
  sia k l'oggetto in R con massimo rapporto valore/peso, cioè, $\frac{v_k}{p_k} = \max\{\frac{v_i}{p_i} | i \in R\}$
  $SOL[k] \leftarrow \min\{\frac{C\text{–}peso}{p_k}, 1\}$
  $peso \leftarrow peso + SOL[k] \cdot p_k$
  $R \leftarrow R\text{–}\{k\}$
ENDWHILE
OUTPUT SOL.

Adesso dimostreremo che l'algoritmo è corretto per il problema ZAINO FRAZIONARIO. Procederemo come al solito seguendo lo schema generale. Osserviamo innanzitutto che l'algoritmo termina in quanto ad ogni iterazione del WHILE viene tolto un oggetto da R e dopo n iterazioni la condizione del WHILE risulta falsa. Quindi il WHILE esegue esattamente n iterazioni.

Sia R_h il valore di R al termine della h-esima iterazione e sia R_0 il valore iniziale. Per ogni $h = 0, 1, \ldots, n$, sia $F_h = \{1, 2, \ldots, n\}\text{–}R_h$. Chiaramente, F_h è l'insieme degli oggetti per cui l'algoritmo ha deciso la frazione, cioè il valore di SOL, nelle prime h iterazioni. Inoltre, risulta $F_n = \{1, 2, ..., n\}$. Proviamo il seguente fatto:

per ogni $h = 0, 1, \ldots, n$ esiste una soluzione ottima SOL^* tale che $SOL^*[i] = SOL[i]$ per ogni i in F_h.

Dimostrazione. La dimostrazione è come al solito per induzione su h. Per $h = 0$ l'enunciato è banalmente vero. Sia ora $h > 0$. Assumendolo vero per h–1 vogliamo dimostrare che è vero anche per h. Sia k l'oggetto considerato nella h-esima iterazione. Se $SOL^*[k] = SOL[k]$ allora abbiamo fatto. In caso contrario deve aversi $SOL^*[k] < SOL[k]$ in quanto la frazione dell'oggetto k presa dall'algoritmo è la massima possibile compatibilmente con le scelte fatte per gli oggetti in F_{h-1}, scelte che per ipotesi induttiva coincidono con quelle fatte in SOL^*.
Da quanto appena provato e dall'ipotesi induttiva deduciamo che

$$\sum_{i \in F_h} SOL[i] \cdot p_i = \sum_{i \in F_h} SOL^*[i] \cdot p_i + (SOL[k]\text{–}SOL^*[k]) \cdot p_k.$$

Inoltre per SOL^* in quanto soluzione ottima deve valere

$$\sum_{i \in F_n} SOL^*[i] \cdot p_i = \min\left\{C, \sum_{i \in F_n} p_i\right\}.$$

Da queste due equazioni si deduce che il peso totale delle frazioni di oggetti in R_h prese da SOL^* è almeno $(SOL[k]\text{–}SOL^*[k]) \cdot p_k$. Infatti, risulta che:

$$\begin{array}{rcl}
\sum_{i \in R_h} SOL^*[i] \cdot p_i & = & \sum_{i \in F_n} SOL^*[i] \cdot p_i \text{–} \sum_{i \in F_h} SOL^*[i] \cdot p_i \\
& = & \min\left\{C, \sum_{i \in F_n} p_i\right\} - \sum_{i \in F_h} SOL^*[i] \cdot p_i \\
& = & \min\left\{C, \sum i \in F_n p_i\right\} - \left(\sum_{i \in F_h} SOL[i] \cdot p_i \text{–} (SOL[k]\text{–}SOL^*[k]) \cdot p_k\right) \\
& = & (SOL[k]\text{–}SOL^*[k]) \cdot p_k + \min\left\{C, \sum_{i \in F n} p_i\right\} - \sum_{i \in F_h} SOL[i] \cdot p_i \\
& \geq & (SOL[k]\text{–}SOL^*[k]) \cdot p_k
\end{array}$$

dove l'ultima disequazione deriva dal fatto che $\sum_{i \in F_h} SOL[i] \cdot p_i \leq \min\left\{C, \sum_{i \in F_n} p_i\right\}$. Si può quindi trovare per ogni oggetto j in R_h un reale c_j con $0 \leq c_j \leq SOL^*[j]$ tale che

$$\sum_{j \in R_h} c_j \cdot p_j = (SOL[k]\text{–}SOL^*[k]) \cdot p_k.$$

Definiamo quindi una soluzione $SOL^\#$ come segue:

$$\begin{array}{lcll}
SOL^\#[j] & = & SOL^*[j] & \text{per } j \text{ in } F_{h-1} \\
SOL^\#[k] & = & SOL[k] & \\
SOL^\#[j] & = & SOL^*[j]\text{–}c_j & \text{per } j \text{ in } R_h
\end{array}$$

Resta solo da dimostrare che il valore della nuova soluzione $SOL^\#$ è ottimo. La variazione di valore tra $SOL^\#$ e SOL^* è data dalla quantità

$$(SOL[k]\text{–}SOL^*[k]) \cdot v_k \text{–} \sum_{j \in R_h} c_j \cdot v_j$$

,
basterà quindi dimostrare che $\sum_{j \in R_h} c_j \cdot v_j \leq (SOL[k]–SOL^*[k]) \cdot v_k$.
In base all'ordine con cui gli oggetti vengono considerati si ha che

$$\frac{v_j}{p_j} \leq \frac{v_k}{p_k} \text{ per ogni } j \in R_h$$

e questo implica che

$$\begin{array}{rcl} \sum_{j \in R_h} c_j \cdot v_j & = & \sum_{j \in R_h} c_j \cdot (\frac{v_j}{p_j}) \cdot p_j \\ & \leq & \sum_{j \in R_h} c_j \cdot (\frac{v_k}{p_k}) \cdot p_j \\ & = & \frac{v_k}{p_k} \sum_{j \in R_h} c_j \cdot p_j \\ & = & \frac{v_k}{p_k} (SOL[k]–SOL^*[k]) \cdot p_k \\ & = & (SOL[k]–SOL^*[k]) \cdot v_k \end{array}$$

Questo completa la dimostrazione

Proviamo ora a valutare il rapporto d'approssimazione dell'algoritmo ZAINO. Sia X una qualsiasi istanza del problema ZAINO. Cerchiamo un limite superiore al valore di una soluzione ottima per l'istanza X, $OTT_{ZAINO}(X)$, in funzione del valore $ZAINO(X)$ della soluzione SOL prodotta dall'algoritmo. Risulta utile per questo scopo considerare anche la soluzione SOL_f prodotta dall'algoritmo ZAINO_FRAZ, sempre sull'istanza X. L'algoritmo ZAINO_FRAZ, di ogni oggetto considerato, seleziona la frazione massima che non comporta il superamento della capacità. L'algoritmo ZAINO, degli oggetti considerati seleziona solo quelli il cui peso non comporta il superamento della capacità. Entrambi gli algoritmi considerano gli oggetti in ordine di rapporto valore/peso non crescente. Stando così le cose, in SOL^f saranno selezionati per intero una serie di oggetti ed eventualmente di un unico oggetto ne sarà selezionata soltanto una frazione. Inoltre la serie di oggetti selezionati per intero in SOL^f sarà selezionata anche in SOL ed il contributo al valore di SOL^f da parte dell'eventuale oggetto selezionato solo in parte sarà certamente inferiore a $vmax(X)$, dove $vmax(X)$ è il massimo fra i valori degli n oggetti di X. Quindi, si ha che

$$V(SOL^f) < ZAINO(X) + vmax(X)$$

dove $V(SOL^f)$ è il valore della soluzione SOL^f. Sappiamo che l'algoritmo ZAINO_FRAZ è corretto per il problema ZAINO FRAZIONARIO, quindi $V(SOL^f) = OTT_{\text{ZAINO FRAZIONARIO}}(X)$. Inoltre, è chiaro che

$$OTT_{ZAINO}(X) \leq OTT_{\text{ZAINO FRAZIONARIO}}(X).$$

Possiamo concludere che $OTT_{ZAINO}(X) < ZAINO(X)+vmax(X)$ e considerando il rapporto di approssimazione si ottiene:

$$\frac{OTT_{ZAINO}(X)}{ZAINO(X)} < 1 + \frac{vmax(X)}{ZAINO(X)}.$$

Qualora un oggetto di valore $vmax(X)$ non sia fra quelli selezionati in SOL il rapporto d'approssimazione potrebbe essere arbitrariamente grande come l'esempio dato precedentemente aveva già mostrato. Un modo ovvio di limitare il rapporto d'approssimazione è quello di considerare un algoritmo che produce la soluzione più conveniente fra la soluzione prodotta dall'algoritmo ZAINO e quella costituita da un unico oggetto di valore massimo. In altre parole, l'algoritmo che chiameremo ZAINO2, produce una soluzione il cui valore, ZAINO2(X), è pari a $\max\{ZAINO(X), vmax(X)\}$. Infatti, calcolando il rapporto di approssimazione per ZAINO2 si ha:

$$\frac{OTT_{ZAINO}(X)}{ZAINO2(X)} < \frac{ZAINO(X) + vmax(X)}{ZAINO2(X)} \leq 2,$$

dato che $ZAINO(X) \leq ZAINO2(X)$ e $vmax(X) \leq ZAINO2(X)$. Dunque, con questa semplice modifica dell'algoritmo ZAINO si ottiene un algoritmo che approssima il problema ZAINO entro un fattore 2.

7. Il Problema RICOPRIMENTO TRAMITE NODI

Una azienda molto grande ha bisogno di assumere nuovi dipendenti per ricoprire un determinato insieme di mansioni. Siccome un dipendente potrebbe ricoprire più mansioni, l'azienda vuole assumere fra i possibili candidati un numero minimo di nuovi dipendenti garantendo che ogni mansione trovi nei nuovi dipendenti qualcuno capace di svolgerla. Una formulazione astratta e generale del problema, che chiameremo COPERTURA, è la seguente. Dato un insieme finito M (l'insieme delle mansioni) e una famiglia $C = \{C_1, C_2, \ldots, C_k\}$ di sottoinsiemi di M (C_i rappresenta le mansioni che l'i-esimo candidato è capace di svolgere) si vuole selezionare una sottofamiglia di C che copre M di cardinalità minima. Formalmente, si vuole trovare un sottoinsieme S di $\{1, 2, \ldots, k\}$ tale che

$$\cup_{i \in SC_i} = M \qquad e \qquad |S| = \min\{|T| | T \text{ è un sottoinsieme di } \{1, 2, \ldots, k\} \text{ e } \cup_{i \in TC_i} = M\}.$$

L'azienda ha necessità di risolvere anche un'altra questione. Allo scopo di coordinare al meglio il lavoro dei suoi dipendenti vuole istituire una commissione composta da una rappresentanza dei dipendenti dell'azienda che supervisioni la distribuzione del lavoro. Per fare ciò conviene che ogni mansione sia in qualche modo rappresentata in seno alla commissione da almeno un dipendente capace di svolgere tale mansione. Però, al contempo, l'azienda vuole minimizzare il numero dei componenti della commissione. Una formulazione astratta e generale del problema, che chiameremo RAPPRESENTANZA, è la seguente. Dato un insieme finito D (l'insieme dei dipendenti) e una famiglia $F = \{F_1, F_2, \ldots, F_m\}$ di sottoinsiemi di D (F_i rappresenta l'insieme dei dipendenti che è capace di svolgere la i-esima mansione) si vuole scegliere un sottoinsieme di D che abbia intersezione non vuota con ogni F_i di minima cardinalità. Formalmente, si vuole trovare un sottoinsieme R di D tale che

$$\text{per ogni } i, R \text{ ha intersezione non vuota con } F_i \qquad e \qquad |R| = \min\{|H| | H \subseteq D \text{ e per ogni } i, H \cap F_i \neq \emptyset\ .$$

In realtà i due problemi sono essenzialmente lo stesso problema. Infatti, è facile vedere come riformulare l'uno nei termini dell'altro. Sia (M, C) una qualsiasi istanza del problema COPERTURA. Costruiamo una istanza (D', E') del problema RAPPRESENTANZA come segue:

$$D' = C \qquad e \qquad F' = \{F'_x | x \in Med F'_x = \{C_i | x \in C_i\}\}$$

La riformulazione interpretata nella "terminologia dell'azienda", diventa: come insieme dei dipendenti D' prendiamo l'insieme dei candidati e così vien da sè che F'_i è l'insieme dei candidati in grado di svolgere la i-esima mansione. Se S è una soluzione ammissibile per l'istanza (M, C) del problema COPERTURA allora S può essere direttamente interpretata anche come soluzione ammissibile per l'istanza costruita (D', F'). E viceversa, ogni soluzione ammissibile R per (D', F') può essere direttamente interpretata come soluzione ammissibile per (M, C). Siccome il valore o misura di una soluzione è per entrambi i problemi la cardinalità, allora questa corrispondenza fra soluzioni ammissibili trasforma ogni soluzione ottima per un problema in una soluzione ottima per l'altro e viceversa. Quindi trovare una soluzione ottima per l'istanza (M, C) è esattamente la stessa cosa che trovare una soluzione ottima per (D', F').

In modo analogo è possibile definire per ogni istanza del problema RAPPRESENTANZA un istanza equivalente del problema COPERTURA. Sia (D, F) una qualsiasi istanza del problema RAPPRESENTANZA. Costruiamo una istanza (M', C') del problema COPERTURA come segue:

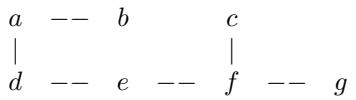
$$M' = E \qquad e \qquad C' = \{C'_y | y \in De C'_y = \{F_i | y \in F_i\}\}.$$

Nella "terminolgia dell'azienda" diventa: l'insieme delle mansioni M' rimane lo stesso, l'insieme dei candidati diventa l'insieme dei dipendenti D e così C_i è l'insieme delle mansioni che l'i-esimo dipendente è in grado di svolgere. Ancora una volta è facile verificare che c'è una corrispondenza completa fra le soluzioni ammissibili dei due problemi e che quindi tale corrispondenza vale anche per le soluzioni ottime.

Ciò che abbiamo appena discusso è un esempio di un fenomeno non infrequente: problemi che a prima vista appaiono molto diversi possono poi risultare essere del tutto equivalenti. A volte la corrispondenza fra i due problemi vale soltanto in un verso. Ad esempio, come abbiamo già visto, il problema FILE può essere riformulato come un caso speciale del problema ZAINO. Un'altro esempio, come vedremo, è dato dal problema RICOPRIMENTO TRAMITE NODI così definito. Dato un grafo $G = (V, E)$ non diretto si vuole trovare un sottoinsieme dei nodi di G che copre tutti gli archi di G di cardinalità minima. Più precisamente, si vuole trovare un sottoinsieme C di V tale che

- per ogni $\{u, v\}$ in E, u è in C or v è in C (cioè, C è un ricoprimento di G)
- $|C| = \min\{|B| | B \subseteq V \text{ e per ogni } \{u, v\} \in E, u \in B \text{ or } v \in B\}$.

Ad esempio, per il grafo qui sotto disegnato



l'insieme $\{d, c, g\}$ non è un ricoprimento mentre $\{d, f, b\}$ è un ricoprimento ottimo.

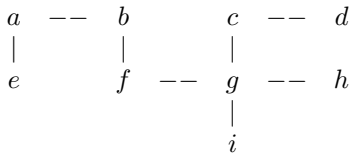
Il problema RICOPRIMENTO TRAMITE NODI è un caso speciale del problema RAPPRESENTANZA. Infatti, corrisponde al caso in cui tutti gli insiemi della famiglia F hanno cardinalità 2.

Pensiamo ad un algoritmo greedy per il problema RICOPRIMENTO TRAMITE NODI. Il primo che viene in mente consiste nello scegliere i nodi in base al numero di archi che coprono. Cioè, si sceglie sempre un nodo che massimizza il

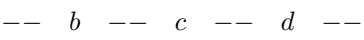
numero di archi che copre fra quelli non coperti dai nodi già scelti. Un descrizione in pseudo codice dell'algoritmo è la seguente.

Algoritmo MAX_GRADO
INPUT un grafo non diretto $G = (V, E)$
$C \leftarrow \emptyset$
WHILE ci sono archi in E non coperti dai nodi in C DO
    Sia u un nodo che massimizza il numero di archi che copre fra quelli non coperti da C
    $C \leftarrow C \cup \{u\}$
ENDWHILE OUTPUT C.

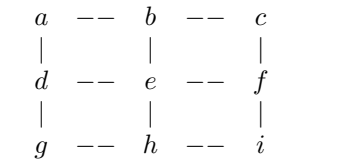
Vediamo come si comporta l'algoritmo su qualche grafo.



Su questo grafo l'algoritmo sceglie nell'ordine i nodi g, b, d, a. Quindi trova una soluzione ottima (perché è ottima?). Consideriamo un'altro grafo ancora più semplice:



Il primo nodo che l'algoritmo sceglie può essere uno qualsiasi fra b, c o d. Però se sceglie c non produrrà una soluzione ottima. Quindi l'algoritmo così com'è non è corretto. Si potrebbe pensare di emendarlo aggiungendo qualche ulteriore criterio di scelta quando ci sono più nodi che massimizzano il numero di nuovi archi che vengono coperti. Ma ciò non è possibile, infatti consideriamo il grafo seguente:



L'algoritmo sceglie il nodo e e poi è costretto a scegliere quattro dei nodi rimanenti. Quindi produce una soluzione di valore 5. Ma la soluzione ottima b, d, f, h ha valore 4.

In realtà l'algoritmo non produce neanche un fattore di approssimazione costante. Per vederlo consideriamo una opportuna famiglia di grafi G_k al variare di un parametro $k \geq 2$. Il grafo $G_k = (V_k, E_k)$ è così definito :

$V_k = S \cup L_2 \cup L_3 \cup \ldots \cup L_k$, dove gli insiemi $S, L_2, L_3, \ldots L_k$ sono disgiunti ed hanno le seguenti cardinalità $|S| = k$, $|L_j| = \left\lfloor \frac{k}{j} \right\rfloor$ per $j = 2, 3, \ldots, k$. $E_k = A_2 \cup A_3 \cup \ldots \cup A_k$ dove, per ogni $j = 2, 3, \ldots, k$, A_j è un insieme di $j \cdot L_j$ archi tali che ognuno ha un estremo in S e l'altro in L_j, ogni nodo in L_j è incidente in esattamente j archi ed ogni nodo in S è incidente in al più uno degli archi di A_j.
Il grafo G_k è tale che i nodi in L_j hanno grado j e i nodi in S hanno grado al più k–1. Quindi il primo nodo che l'algoritmo sceglie è l'unico nodo di L_k (che ha grado k). Dopo questa scelta i nodi in S avranno grado al più k–2. Perciò, l'algoritmo sceglierà tutti i $\left\lfloor \frac{k}{k-1} \right\rfloor$ nodi di L_{k-1} (che hanno grado k–1). Dopo aver scelto tutti i nodi di L_{k-1} i nodi in S avranno grado al più k–3. Quindi, l'algoritmo procederà scegliendo tutti i nodi in L_{k-2}, poi quelli in L_{k-3} e così via fino ad arrivare a scegliere tutti nodi in L_2. Infine l'algoritmo avrà prodotto un ricoprimento composto da tutti i nodi in $L_2 \cup L_3 \cup \ldots \cup L_k$. Ma è chiaro che un ricoprimento è anche dato dall'insieme S. Confrontiamo le cardinalità dei due ricoprimenti. Per la cardinalità del ricoprimento prodotto dall'algoritmo si ha che

$$|L_2 \cup L_3 \cup \ldots \cup L_k| = \sum_{j=2}^{k} |L_j| = \sum_{j=2}^{k} \left\lfloor \frac{k}{j} \right\rfloor .$$

Per ottenere un buon limite inferiore all'ultima sommatoria procediamo come segue:

$$\begin{aligned}
\sum_{j=2}^{k} \left\lfloor \frac{k}{j} \right\rfloor &= \sum_{i=1}^{\lfloor \log k \rfloor} \sum_{j>2^{i-1}}^{2^i} \left\lfloor \frac{k}{j} \right\rfloor \\
&\geq \sum_{i=1}^{\lfloor \log k \rfloor} \sum_{j>2^{i-1}}^{2^i} \left\lfloor \frac{k}{2^i} \right\rfloor \\
&= \sum_{i=1}^{\lfloor \log k \rfloor} 2^{i-1} \left\lfloor \frac{k}{2^i} \right\rfloor \\
&\geq \sum_{i=1}^{\lfloor \log k \rfloor} 2^{i-1} \left\lfloor \frac{2^{\lfloor \log k \rfloor}}{2^i} \right\rfloor \\
&= \sum_{i=1}^{\lfloor \log k \rfloor} 2^{i-1} \left\lfloor 2^{\lfloor \log k \rfloor - i} \right\rfloor \\
&= \sum_{i=1}^{\lfloor \log k \rfloor} 2^{i-1} 2^{\lfloor \log k \rfloor - i} \\
&= \sum_{i=1}^{\lfloor \log k \rfloor} 2^{\lfloor \log k \rfloor - 1} \\
&= 2^{\lfloor \log k \rfloor - 1} \lfloor \log k \rfloor \\
&\geq 2^{\log k - 2} \lfloor \log k \rfloor \\
&= k \cdot \frac{\lfloor \log k \rfloor}{4}
\end{aligned}$$

Quindi, il ricoprimento prodotto dall'algoritmo MAX_GRADO su input G_k ha cardinalità maggiore od uguale a $k \cdot \frac{\lfloor \log k \rfloor}{4}$. Mentre esiste un ricoprimento di G_k di cardinalità k. Perciò il rapporto di approssimazione è maggiore od uguale a

$$\frac{\lfloor \log k \rfloor}{4}.$$

Dunque, al crescere di k si può avere un rapporto di approssimazione che è peggiore di una qualsivoglia costante.

Un algoritmo che intuitivamente sembrava avere le carte in regola per risolvere il problema o perlomeno per fornire una buona approssimazione ha disatteso le nostre aspettative. Eppure l'algoritmo MAX_GRADO ad ogni passo fa la scelta che in quella situazione sembra proprio essere la migliore. Però, i grafi G_k mostrano che la scelta che localmente sembra essere la migliore in effetti può essere molto lontana dall'essere anche soltanto una buona scelta. In realtà ben difficilmente l'algoritmo avrebbe potuto essere corretto perché RICOPRIMENTO TRAMITE NODI è un problema NP-completo. Ciò non toglie che potrebbero comunque esistere algoritmi efficienti che garantiscono una buona approssimazione. Uno di questi si basa su una semplice osservazione. Se in un grafo G ci sono m archi fra loro disgiunti allora un qualsiasi ricoprimento di G deve avere almeno m nodi. Questo è ovvio, un qualsiasi ricoprimento deve coprire gli m archi ma essendo essi disgiunti occorrono almeno m nodi per coprirli. Un algoritmo che costruisce un ricoprimento in modo strettamente legato alla contemporanea costruzione di un insieme di archi disgiunti produrrebbe un ricoprimento che non può essere troppo lontano da un ricoprimento ottimo. L'idea allora è molto semplice. Ad ogni passo si sceglie un arco fra quelli non ancora coperti ed entrambi gli estremi sono aggiunti all'insieme di copertura. Così facendo si garantisce che tutti gli archi scelti sono fra loro disgiunti. La descrizione in pseudo codice dell'algoritmo è immediata.

Algoritmo ARCHI_DISGIUNTI INPUT un grafo non diretto $G = (V, E)$
$C \leftarrow \emptyset$
WHILE ci sono archi in E non coperti da nodi in C DO
    Sia $\{u, v\}$ un arco in E non coperto da nodi in C
    $C \leftarrow C \cup \{u\} \cup \{v\}$
ENDWHILE
OUTPUT C.

Da quanto sopra detto è chiaro che l'algoritmo produce sempre un ricoprimento di cardinalità al più doppia rispetto a quella di un ricoprimento ottimo. Quindi l'algoritmo ARCHI_DISGIUNTI garantisce un fattore di approssimazione pari a 2. Non si conoscono a tutt'oggi algoritmi efficienti che garantiscano un fattore di approssimazione migliore di 2. Per i problemi da cui eravamo partiti, cioè COPERTURA e RAPPRESENTANZA, la situazione è anche peggiore. Non si conoscono algoritmi efficienti che garantiscano un fattore di approssimazione migliore di un fattore che è logaritmico nella dimensione dell'istanza. Però si può dimostrare che l'adattamento dell'algoritmo MAX_GRADO per i due problemi suddetti garantisce un fattore di approssimazione che è logaritmico nella dimensione dell'istanza. Quindi alla fine quell'algoritmo aveva delle buone qualità.

8. Il Problema RIPARTIZIONE

Vediamo ora un problema che può essere visto come uno degli esempi più semplici di problemi di pianificazione (scheduling). Ovvero problemi che richiedono di pianificare certe attività allo scopo di minimizzare l'utilizzo di un qualche tipo di risorsa.

Il problema RIPARTIZIONE è così definito. Dati n lavori con tempi di esecuzione $t_1, t_2, \ldots, t_n$, si vogliono ripartire questi lavori tra due operai in modo da minimizzare il tempo di esecuzione di tutti i lavori. In altre parole, si vuole la bipartizione S dei lavori per cui risulta minima la quantità

$$\max \left\{ \sum_{i \text{ per cui } S[i]=1} t_i, \sum_{i \text{ per cui } S[i]=2} t_i \right\},$$

dove $S[i] = 1$ se il lavoro i è assegnato al primo operaio ed $S[i] = 2$ se è invece assegnato al secondo.

Ad esempio per la semplice istanza

	1	2	3	4
t	2	4	5	6

la bipartizione $(1, 2, 2, 1)$ di valore 9 è ottima.

Non è noto alcun algoritmo efficiente che risolve il problema per ogni istanza, tuttavia è immediato vedere che un qualsiasi algoritmo di ripartizione fornisce un fattore di approssimazione al più 2. Infatti, sia ALG il valore della soluzione prodotta dall'algoritmo, OTT il valore della soluzione ottima e T la somma dei tempi degli n lavori. Il tempo massimo d'esecuzione si ha quando tutti i lavori vengono assegnati ad un singolo operaio, quindi necessariamente $ALG \leq T$, d'altra parte il meglio che ci si può aspettare è di riuscire a bipartire i lavori in modo equo tra i due operai (i.e. ad entrambi gli operai occorrerà lo stesso tempo per eseguire i lavori ad essi assegnati), quindi $\frac{T}{2} \leq OTT$. Quindi:

$$\frac{ALG}{OTT} \leq 2.$$

Cerchiamo un semplice algoritmo greedy che ci permetta di migliorare questo fattore di approssimazione. Una prima idea è la seguente. L'algoritmo assegna i lavori al primo operaio fino a che questo risulta oberato di un tempo di lavoro non superiore a $T/2$, il primo lavoro che viola questa condizione viene per il momento accantonato ed i rimanenti lavori vengono assegnati al secondo operaio. Il lavoro accantonato viene infine assegnato all'operaio che risulta meno oberato. In pseudo codice:

Algoritmo RIPARTIZIONE1
INPUT i tempi $t_1, t_2, \ldots, t_n$ di n lavori
sia T la somma dei tempi degli n lavori
$tempo1 \leftarrow 0$
$j \leftarrow 1$
WHILE $tempo1 + t_j \leq T/2$ DO
    $SOL[j] \leftarrow 1$
    $tempo1 \leftarrow tempo1 + t_j$
    $j \leftarrow j + 1$
ENDWHILE
$tempo2 \leftarrow 0$
FOR $k \leftarrow j + 1$ TO n DO
    $SOL[k] \leftarrow 2$
    $tempo2 \leftarrow tempo2 + t_k$
ENDFOR
IF $tempo1 \leq tempo2$ THEN $SOL[k] \leftarrow 1$
ELSE $SOL[k] \leftarrow 2$
OUTPUT SOL.

Mostreremo ora che

l'algoritmo RIPARTIZIONE1 ha un rapporto di approssimazione limitato da $3/2$.
Dimostrazione. Sia i l'indice del lavoro "accantonato". Deve quindi aversi $tempo1 \leq T/2$, $tempo1 + t_i > T/2$ e $tempo2 < T/2$. Quindi prima dell'assegnamento del lavoro di indice i entrambi gli operai lavorano per un tempo al più $T/2$. Più precisamente esistono $a \geq 0$ e $b > 0$ tali che

$$tempo1 + a = \tfrac{T}{2}, \quad tempo2 + b = \tfrac{T}{2} \quad e \quad a + b = t_i$$

Distinguiamo due casi.

(1) $a \geq b$: il lavoro di indice i verrà assegnato al primo operaio cosicchè il secondo operaio avrà un carico di lavoro inferiore a $T/2$ mentre il tempo di lavoro assegnato al primo operaio sarà

$$tempo1 + t_i = \frac{T}{2} - a + (a + b) = \frac{T}{2} + b \leq \frac{T}{2} + \frac{t_i}{2}.$$

(2) $a < b$: il lavoro di indice i verrà assegnato al secondo operaio cosicchè il primo operaio avrà un carico di lavoro al più $T/2$ mentre il tempo di lavoro assegnato al secondo operaio sarà

$$tempo2 + t_i = \frac{T}{2} - b + (a + b) = \frac{T}{2} + a < \frac{T}{2} + \frac{t_i}{2}.$$

In entrambi i casi risulta quindi $V(SOL) \leq T/2 + t_i/2$, dove $V(SOL)$ è il valore della soluzione prodotta dall'algoritmo. Ricordando ora che in ogni caso deve aversi $T/2 \leq OTT$ e che $t_i \leq OTT$ otteniamo

$$V(SOL) \leq OTT + \frac{OTT}{2} = \frac{3}{2} \cdot OTT.$$

È facile provare che la limitazione del rapporto di approssimazione appena ottenuta non può essere migliorata, in quanto non è difficile produrre istanze che producono esattamente quel rapporto. Si consideri ad esempio l'istanza

	1	2	3
t	1	2	1

l'algoritmo produce la ripartizione $(1, 1, 2)$ di valore 3 mentre la soluzione ottima è $(1, 2, 1)$ di valore 2.

Vediamo ora un diverso algoritmo che ci permette di ottenere un migliore fattore di approssimazione in cambio di un maggior tempo di esecuzione. Una buona idea da cui partire è quella di pensare di assegnare i lavori in modo da mantenere il più possibile bilanciato il carico di lavoro dei due operai. In altri termini ogni nuovo lavoro considerato verrà assegnato all'operaio cui è stato assegnato un carico di lavoro minore. Questo accorgimento da solo non basta, si consideri ad esempio l'istanza

	1	2	3	4	5
t	1	1	1	1	4

L'algoritmo produrrebbe la ripartizione $(1, 2, 1, 2, 1)$ di valore 6 mentre la soluzione ottima $(1, 1, 1, 1, 2)$ ha valore 4. Questo già dice che il rapporto di approssimazione non può essere inferiore a $3/2$.

Qual'è il punto debole di questo approccio? È facile convincersi che la presenza di un lavoro con un lungo tempo di esecuzione è in grado di sbilanciare di molto il bilanciamento che la strategia proposta tenta di mantenere. Se poi questo lavoro compare tra gli ultimi lavori ancora da assegnare lo sbilanciamento può difficilmente essere riassorbito dall'assegnamento dei pochi lavori restanti. Alla luce di questa osservazione può quindi sembrare una buona idea procedere all'assegnazione dei lavori per tempi d'esecuzione decrescenti. Intuitivamente avere a disposizione nella fase finale dell'assegnamento lavori con tempi di esecuzione brevi permette di riassorbire eventuali sbilanciamenti dovuti agli assegnamenti fatti in precedenza. Lo pseudo codice del nuovo algoritmo proposto è quindi il seguente.

```
Algoritmo RIPARTIZIONE2
INPUT i tempi t_1, t_2, ..., t_n di n lavori
F ← {1, 2, ..., n}
tempo1 ← 0
tempo2 ← 0
WHILE F ≠ ∅ DO
    Sia i un lavoro con tempo di esecuzione massimo in F, cioè, t_i = max{t_k | k ∈ F}
    IF tempo1 ≤ tempo2 THEN
        SOL[i] ← 1
        tempo1 ← tempo1 + t_i
    ELSE
        SOL[i] ← 2
        tempo2 ← tempo2 + t_i
    ENDIF
    F ← F−{i}
ENDWHILE
```

OUTPUT *SOL*.

Una implementazione efficiente può essere data sulla falsa riga di quella per l'algoritmo TERMINA_PRIMA. ordinando i compiti preliminarmente in ordine non decrescente di tempo di esecuzione e poi distribuendoli ai due operai seguendo quest'ordine. Chiaramente, la complessità è $O(n \log n)$.

Mostreremo ora che

l'algoritmo RIPARTIZIONE2 ha un rapporto di approssimazione limitato da 7/6.

Dimostrazione. È facile vedere che quando i lavori da essegnare sono al più 4 allora l'algoritmo non sbaglia. Si assuma quindi che per l'istanza in esame ci sono almeno 5 lavori da assegnare e sia i il lavoro che richiede il minimo tempo di esecuzione. Distinguiamo due casi in funzione dell'operaio che riceve il lavoro di minima durata e faremo vedere che in entrambi i casi il rapporto di approssimazione non supera 7/6. Sia $V(SOL)$ il valore della soluzione prodotta dall'algoritmo e sia T la somma dei tempi di esecuzione.

(1) **Il lavoro di durata minima è assegnato all'operaio che lavora di più.** In questo caso, tolto il lavoro i, ad entrambi gli operai sono assegnati lavori per un tempo di almeno $V(SOL)$–t_i. Deve quindi aversi $2(V(SOL)–t_i) \leq T–t_i$, vale a dire $V(SOL) \leq T/2 + t_i/2$. Ricordando poi che $T/2 \leq OTT$ si ha per il rapporto di approssimazione la limitazione

$$\frac{V(SOL)}{OTT} \leq 1 + \frac{t_i}{2 \cdot OTT}$$

Ora se fosse $1 + t_i/(2OTT) > 7/6$ dovrebbe essere $OTT < 3t_i$ e poiché t_i è il lavoro di minima durata ciò implicherebbe che ad entrambi gli operai sono stati assegnati al più due lavori o, in altri termini, che i lavori da assegnare erano al più 4 e questo contraddice l'assunzione che i lavori da essegnare erano almeno 5.

(2) **Il lavoro di durata minima è assegnato all'operaio che lavora di meno.** In questo caso sia k l'ultimo lavoro assegnato all'operaio che lavora di più. Si consideri l'istanza coi soli lavori di durata almeno t_k. Sia OTT' il valore di una soluzione ottima per questa nuova istanza e SOL' la soluzione prodotta dall'algoritmo RIPARTIZIONE2 su questa nuova istanza. Ovviamente $OTT' \leq OTT$ in quanto si tratta ora di bipartire un sottoinsieme degli n lavori che si avevano originariamente. Inoltre, poiché l'algoritmo RIPARTIZIONE2 assegna i lavori in ordine di tempo di esecuzione decrescente, l'algoritmo assegna questi lavori allo stesso modo sia per la nuova istanza che per la vecchia, e quindi $V(SOL') = V(SOL)$. Deve quindi aversi $V(SOL)/OTT \leq V(SOL')/OTT'$. Basterà quindi mostrare che $V(SOL')/OTT' \leq 7/6$, ma questo segue immediatamente notando che nella nuova istanza il lavoro di durata minima è assegnato all'operaio che lavora di più, caso per il quale abbiamo gia dimostrato che il rapporto di approssimazione non supera 7/6.

È facile mostrare che la limitazione del rapporto di approssimazione per RIPARTIZIONE2 appena ottenuta non può essere migliorata, in quanto esistono istanze molto semplici che producono esattamente quel rapporto. Si consideri ad esempio l'istanza

	1	2	3	4	5
t	3	3	2	2	2

L'algoritmo produce la ripartizione $(1, 2, 1, 2, 1)$ di valore 7 mentre la soluzione ottima $(1, 1, 2, 2, 2)$ ha valore 6.