Teoria della computazione

Appunti originali di Federica Adobbati (AA 2016/2017)
Revisione completa di Luca Chiodini (AA 2018/2019)

Università degli Studi di Milano-Bicocca
E-mail: {f.adobbati, l.chiodini}@campus.unimib.it

Obiettivo: Il principale obiettivo della teoria della computazione è capire ciò che può essere calcolato in modo automatico. Ci si chiede, ad esempio, quali verità logiche possono essere dimostrate.

Il primo risultato in questo senso fu dato da Turing, che utilizzò una macchina astratta per dimostrare che non tutte le verità logiche possono essere dimostrate.

1 Teoria della complessità computazionale

La teoria della complessità computazionale è una branca della teoria della computazione che ha come obiettivo principale la classificazione dei problemi sulla base della loro complessità, definita mediante tempo e spazio richiesti per risolverli. I problemi classificati sono quelli di **decisione**. Tali problemi possono essere descritti attraverso funzioni binarie che prendono in input una stringa costituita dai simboli 0 e 1 e restituiscono un valore di verità (rappresentabile come un bit, 0 oppure 1). Chiameremo π un problema di decisione, che dato un input x restituisce un valore booleano, e f_π la funzione associata a π definita come:

$$f_\pi : \{0, 1\}^* \to \{0, 1\},$$

dove * indica l'arbitrarietà della lunghezza della stringa di input.

2 La macchina di Turing

2.1 Introduzione informale

La macchina di Turing è una macchina astratta costituita da un nastro infinito e da una testina in grado di leggere e scrivere il contenuto della cella del nastro su cui punta. Inoltre è definito un controllo, ossia un programma che ne descrive il funzionamento. Il controllo può essere rappresentato graficamente attraverso dei cerchi per indicare gli stati del programma e degli archi orientati che indicano la possibilità di passaggio da uno stato all'altro; uno stato finale viene indicato con due circonferenze concentriche. Gli stati indicano quali regole devono essere applicate, le transizioni (indicate con le frecce) sono regole associate agli stati. Lo stato finale in cui la macchina si arresta è detto di **halt**.

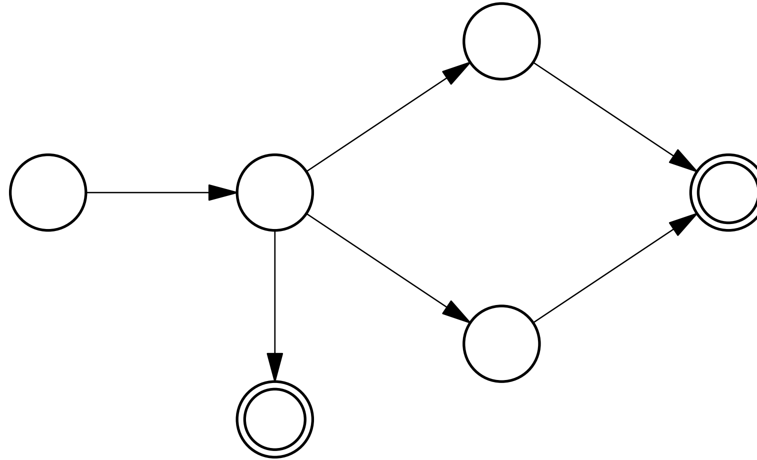


Fig. 1. Esempio di grafo per la descrizione del controllo di una TM.

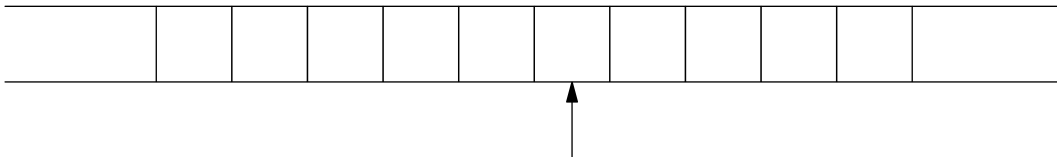


Fig. 2. Nastro (e testina) di una macchina di Turing

Non tutte le macchine di Turing sono costruite in questo modo, ad esempio alcune hanno nastro infinito solo in una direzione o hanno più nastri; nel seguito comunque ci limiteremo a considerare il modello descritto sopra.

Nel nostro modello, in ogni istante di tempo la testina esegue le seguenti operazioni:

1. legge un simbolo;
2. scrive un simbolo;
3. si sposta di una posizione a sinistra o a destra.

Inizialmente la testina è posizionata sul primo simbolo della stringa in input e ad ogni istante di tempo si eseguono i tre punti descritti sopra.

Esempio 1. Tenendo conto che consideriamo vuote (simbolo di blank) le celle in cui non compare alcun carattere, partiamo dalla seguente configurazione:

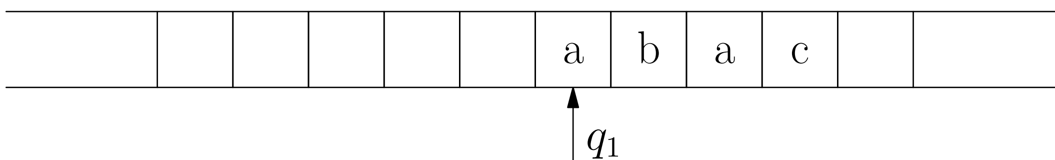


Fig. 3. Nastro nell'istante di tempo 1

Siamo dunque nello stato q_1 e leggiamo a. Andiamo a cercare nel grafo la regola che descrive il comportamento della macchina in questa situazione; supponiamo di trovare:

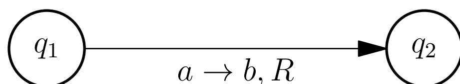


Fig. 4. Una parte del grafo di controllo della TM che descrive una transizione.

Questo ci indica che se nello stato q_1 leggiamo a, allora scriviamo b, muoviamo la testina a destra e passiamo nello stato q_2. Se avessimo trovato indicata una L, naturalmente ci saremmo mossi a sinistra. Ci troveremo dunque all'istante di tempo 2 in questa situazione:

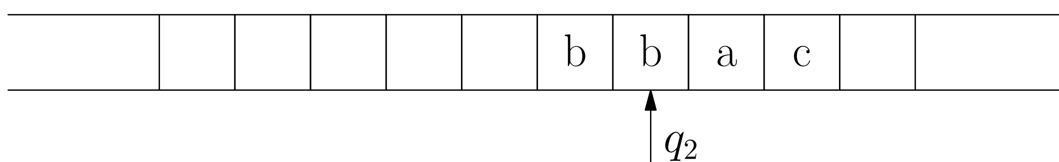


Fig. 5. Nastro nell'istante di tempo 2

Nella formulazione classica, le macchine di Turing sono automi di tipo deterministico, ossia preso un certo stato ed effettuata la lettura di un certo carattere, deve esserci una ed una sola transizione. Non è permesso ad esempio che dallo stato q_1 partano due frecce che abbiano come ipotesi la lettura di a. Non ci sono problemi invece se da uno stesso stato partono più frecce con simboli letti differenti.

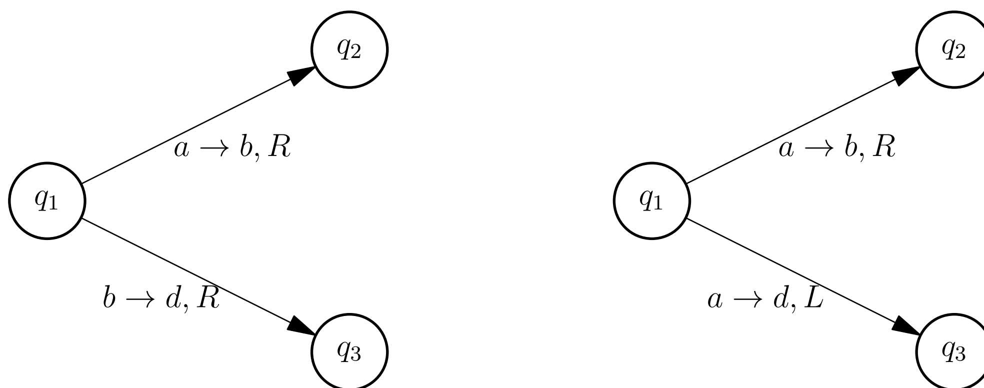


Fig. 6. La situazione a sinistra è permessa, quella a destra **non** lo è.

Gli stati finali non possono avere transizioni uscenti. È lecita la situazione in cui in un certo stato non siano definite transizioni per ogni simbolo: se la macchina non

trova alcuna transizione da eseguire va in stato di halt. D'altra parte la macchina di Turing **non** accetta l'input se non trova regole per il simbolo (ed è in uno stato non finale) oppure se va in loop. Lo accetta se entra in halt trovandosi in uno stato finale.

Questo pone un quesito: cosa si intende per "accettare un linguaggio"?

2.2 Definizione formale di una TM

Definizione 1. Definiamo **funzione di transizione** una funzione δ che prende in input lo stato di partenza e il simbolo che viene letto e restituisce una terna composta dallo stato di arrivo, il simbolo da scrivere e l'indicazione del movimento. Scriveremo ad esempio

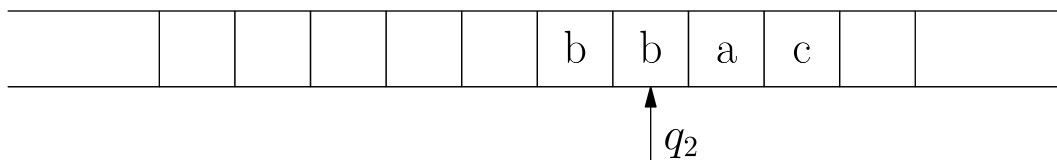
$$\delta(q_1, a) = (q_2, b, R)$$

Indichiamo con M la macchina di Turing e avremo che $M = (Q, \Sigma, \Gamma, \delta, q_0, \diamondsuit, F)$, dove Q è l'insieme degli stati, Σ è l'alfabeto dei simboli in input, Γ è l'alfabeto dei simboli del nastro, δ è la funzione definita sopra, q_0 è lo stato iniziale, $\diamondsuit$ rappresenta il blank e F è l'insieme degli stati finali.

La configurazione di una macchina di Turing si può descrivere ad ogni istante convenzionalmente come:

caratteri a sinistra del cursore - stato - caratteri dal cursore in avanti.

Ad esempio nella situazione:



la macchina in quell'istante viene descritta con la scrittura bq_2bac.

Per indicare una mossa, ovvero il passaggio da una configurazione all'altra, inseriremo una **f** fra le due. Riconsiderando il caso dell'Esempio (1) scriveremo

$$q_1abac \ f \ bq_2bac.$$

Analogamente è possibile concatenare fra loro un numero di mosse arbitrario. Se vi sono due o più configurazioni, è possibile abbreviare la scrittura indicando solo la configurazione iniziale e quella finale separate da f^* (che significa appunto un numero arbitrario di mosse).

3 Funzioni calcolabili

Definizione 2. Una funzione $f : D \to S$ è una relazione che associa ad ogni elemento del dominio D uno ed un solo elemento del codominio S. Preso $w \in D$, si scriverà che $f(w) = z \in S$.

Una funzione può avere più parametri, ad esempio la funzione $f(x, y) = x + y$ ne ha due.

I numeri interi si possono rappresentare in vari modi, ad esempio si può usare la rappresentazione decimale, quella binaria o quella unaria, che consiste nel rappresentare ogni numero con un numero di 1 corrispondente; quindi ad esempio se consideriamo 5 in notazione decimale, con la notazione unaria scriveremo 11111, cioè scriveremo per 5 volte 1. La notazione unaria è la più facile da utilizzare con le macchine di Turing.

Una funzione f è calcolabile se esiste una macchina di Turing M tale che, avendo come configurazione iniziale $q_0\ w$, termina in uno stato q_f con $f(w)$ sul nastro (e questo deve accadere per ogni possibile input $w \in D$). Più formalmente possiamo dare la seguente definizione.

Definizione 3. Una funzione $f : D \to S$ è **calcolabile** se esiste una macchina di Turing M tale che

$$q_0 w \succ^* q_f f(w),$$

per ogni $w \in D$ ($\succ^*$ indica una sequenza di lunghezza arbitraria di configurazioni).

Esempio 2. La funzione $f(x, y) = x + y$, con x, y interi è calcolabile. Infatti la macchina di Turing rappresentata nella figura seguente prende come stringa in input $x0y$ e restituisce come stringa in output $xy0$:

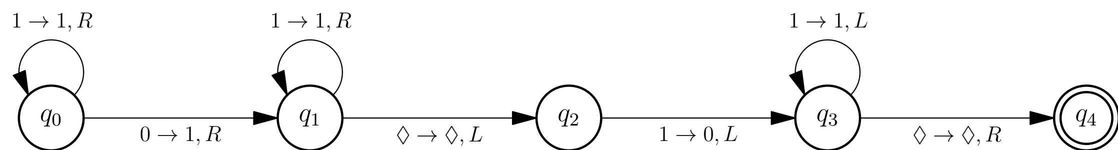


Fig. 7. Una macchina di Turing che calcola $x + y$.

Esempio 3. Un altro esempio di funzione calcolabile è $f(x) = 2x$ con x intero. In questo caso la macchina prende in input x in unario e restituisce in output xx sempre in notazione unaria. Un possibile algoritmo per una MdT è il seguente:

- sostituisce ogni 1 con un simbolo $;
- ripete finché non ci sono più simboli $:
  - trova il simbolo $ più a destra e lo sostituisce con 1;
  - trova il primo spazio vuoto sulla destra e inserisce 1.

3.1 Linguaggi accettati e decisi

Definizione 4. Per ogni macchina di Turing M, definisco **linguaggio accettato** dalla macchina M l'insieme di stringhe

$$L(M) = \{w : q_0 w\ f^*\ x_1 q_f x_2\}$$

dove q_0 è lo stato iniziale e q_f uno stato finale (mentre x_1 e x_2 sono sequenze di simboli qualsiasi).

Il concetto di linguaggio accettato può essere descritto anche tramite le seguenti definizioni:

Definizione 5. Un linguaggio L è **accettato** se esiste una macchina di Turing M che computa la funzione f_L così definita:

$$f_L(x) = 1 \text{ se e solo se } x \in L$$

in caso contrario f_L può anche non essere definita.

Definizione 6. Un linguaggio L è **accettato** da una macchina di Turing M se M accetta ogni $x \in L$ e non accetta tutti gli input $x \notin L$; ossia, preso $x \notin L$, M si arresta in halt senza raggiungere uno stato finale oppure entra in loop.

È importante prestare attenzione alla differenza con la definizione di **linguaggio deciso**.

Definizione 7. Un linguaggio L è **deciso** (o ricorsivo) se esiste una macchina di Turing M che computa la funzione f_L in questo modo:

$$f_L(x) = \begin{cases} 1 & x \in L \\ 0 & x \notin L \end{cases}$$

ovvero, M deve terminare anche se l'input non appartiene al linguaggio.

Definizione 8. Una problema P di decisione si dice **decidibile** se esiste una macchina di Turing che *decide* il linguaggio

$$L_P = \{x : P \text{ dà come risultato 1 prendendo in input } x\}$$

Esempio 4 (Problema di corrispondenza di Post).

Date due sequenze di parole $A = \langle v_1, v_2, \dots, v_n\rangle$ e $B = \langle w_1, w_2, \dots, w_m\rangle$, ci chiediamo se esiste una sequenza di $n \geq 1$ indici $\langle i_1, ..., i_n\rangle$ (compresi tra 1 e m), tali che

$$v_{i_1} \cdot v_{i_2} \cdot \ldots \cdot v_{i_n} = w_{i_1} \cdot w_{i_2} \cdot \ldots \cdot w_{i_n}$$

Il simbolo $\cdot$ tra due stringhe indica la concatenazione.

Prendiamo come esempio $A = \langle 1, 10111, 10\rangle$ e $B = \langle 111, 10, 110\rangle$. Una soluzione è la sequenza di indici $\langle 2, 1, 3\rangle$, infatti componendo in questo modo le sequenze otterremmo per entrambe 10111110.

In altre parole, ci stiamo chiedendo se è possibile comporre con i due dizionari A e B una stessa frase. Questo è un classico esempio di problema indecidibile.

Tesi di Turing Qualsiasi calcolo che possa essere eseguito in modo meccanico può essere eseguito da una macchina di Turing. Quindi, un calcolo può essere eseguito meccanicamente se e solo se può essere eseguito da una macchina di Turing; dunque non c'è un modello di computazione più potente della macchina di Turing. Quando diciamo "esiste un algoritmo" sottintendiamo che "esiste una macchina di Turing che esegue un algoritmo".

Definizione 9. Un algoritmo per una funzione $f(w)$ è una macchina di Turing che calcola $f(w)$.

4 Problemi NP

Dato un problema siamo interessati a conoscerne la complessità e i tempi di calcolo di una soluzione. Possiamo estendere la definizione di **linguaggio accettato** inserendo un vincolo di tempo $T(|x|)$, dove $|x|$ è la lunghezza dell'input x e $T : \mathbb{N} \to \mathbb{N}$ è una funzione calcolabile da una macchina di Turing.

Preso un linguaggio di decisione L_π, una macchina di Turing M **accetta** L_π in tempo $T(n)$ se per ogni $x \in L_\pi$, con $|x| = n$, M accetta x in $T(n)$ mosse (o configurazioni).

Definizione 10. Chiamiamo **P** la classe dei linguaggi di decisione accettati da una macchina di Turing in tempo polinomiale, ovvero con un vincolo di tempo $T(n) = c \cdot n^p$ per qualche valore di c costante e $p \in \mathbb{N}$.

Dato un algoritmo $\mathcal{A}$, una definizione equivalente può essere:

Definizione 11. **P** è la classe dei linguaggi di decisione accettati da un algoritmo $\mathcal{A}$ in tempo polinomiale, ovvero $T(n) = c \cdot n^p$.

Ponendo come condizione per accettare il linguaggio il limite di tempo $T(n)$, è possibile definire il complemento del linguaggio L_π:

$$\overline{L_\pi} = \{x : f_\pi(x) = 0\}$$

che è accettato da una macchina di Turing M'. In particolare M' "simula" M sull'input x e se dopo $T(n)$ mosse (dove $n = |x|$), M non ha ancora accettato x, allora M' restituisce 1 ($x \notin L_\pi$).

Consideriamo ora un altro modello per la macchina di Turing, che chiamiamo *macchina di Turing nondeterministica (NDTM)*. In questo modello è permesso, data la lettura di un carattere in certo stato, avere due scelte equivalenti come nell'esempio mostrato nella parte destra della figura (6). La definizione di una NDTM è analoga a quella di una TM deterministica, con la differenza che non conosciamo con certezza le mosse della macchina. Sappiamo quindi, dato uno stato iniziale ed uno finale, che esiste una sequenza di configurazioni che dallo stato iniziale porta allo stato finale, ma non la conosciamo in modo deterministico.

Definizione 12. Sia $T : \mathbb{N} \to \mathbb{N}$ una funzione calcolabile da una TM e L_π un linguaggio di decisione. Una macchina di Turing nondeterministica M **accetta** L_π in tempo $T(n)$ se $\forall x \in L_\pi$, con $|x| = n$, M accetta x in $T(n)$ mosse.

Definizione 13. Chiamiamo **NP** la classe dei linguaggi di decisione accettati da una NDTM in tempo polinomiale $T(n) = c \cdot n^p$, per qualche costante c e $p \in \mathbb{N}$.

Consideriamo un algoritmo $\mathcal{A}$, che prende in input oltre alla stringa x (l'input vero e proprio) una stringa y di lunghezza polinomiale nell'input detta *certificato*. Diciamo che $\mathcal{A}$ accetta L_π in tempo $T(n)$ se $\forall x \in L_\pi$ con $|x| = n$, $\mathcal{A}(x, y)$ termina entro $T(n)$ passi producendo 1. Possiamo definire allora NP come segue:

Definizione 14. Chiamiamo **NP** la classe dei linguaggi di decisione accettati in tempo polinomiale $T(n) = c \cdot n^p$ (con c costante e $p \in \mathbb{N}$) da un algoritmo $\mathcal{A}(x, y)$ (dove y è il certificato).

In altre parole, mentre P è la classe dei problemi *risolvibili* (di cui si può calcolare una soluzione) in tempo polinomiale, NP è la classe dei problemi *verificabili* (avendo un certificato, si verifica se è una soluzione) in tempo polinomiale. Chiaramente P $\subseteq$ NP, nessuno però è ancora riuscito a dimostrare che P = NP oppure che P $\neq$ NP.

4.1 Classici esempi di problemi NP

Presentiamo alcuni classici esempi di problemi che sono in NP (ma ovviamente diversi da quelli banalmente in P).

Travelling salesman problem (TSP) Date n città e la distanza $d(u, v)$ che separa ogni coppia di città u e v, esiste un tour (che visita tutte le città una e una sola volta), con inizio e fine nel nodo u, di lunghezza al più k? Per formalizzare il problema utilizziamo un grafo pesato $G(V, E)$ completo, in cui ogni città è rappresentata da un nodo e ogni coppia di città è collegata da un arco pesato con la distanza. Il cammino trovato deve visitare ogni città una e una sola volta. Questo problema è in NP (ed è anche NP-difficile, come mostreremo più avanti).

Vertex cover Dato un grafo $G(V, E)$ non orientato, esiste una copertura tramite vertici $V' \subseteq V$ tale che $|V'|$ sia al più k? Per copertura intendiamo un sottoinsieme V' di V tale che per ogni arco $(u, v) \in E$ almeno uno tra u e v appartiene a V'.

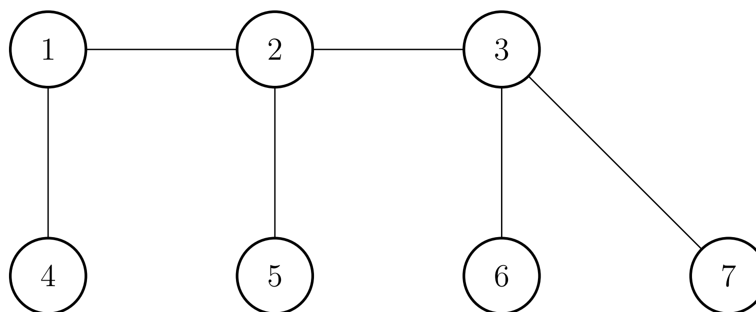


Fig. 8. Esempio di input per VERTEX-COVER

Dato il grafo in figura, possiamo chiederci se i vertici 1 e 2 costituiscono una copertura. La risposta è no: preso ad esempio l'arco $(3, 6)$ nessuno dei due vertici appartiene a V'. Posto $V' = \{1, 2, 6\}$, esso è una copertura? Anche in questo caso la risposta è no, perché l'arco $(3, 7)$ non ha vertici in quell'insieme. $V' = \{1, 2, 6, 7\}$ costituisce una copertura, poiché ogni arco ha almeno un vertice nell'insieme V'. Anche VERTEX-COVER è un problema di decisione in NP, poiché avendo una

possibile soluzione (ad esempio $\{1, 2, 6, 7\}$) si verifica in tempo polinomiale che lo sia effettivamente per il grafo in input. In questo esempio $\{1, 2, 6, 7\}$ costituisce il certificato. Si evince facilmente che esiste un algoritmo polinomiale che, dato un certificato, è in grado di verificare che per ogni arco ci sia almeno un vertice nell'insieme V' e se questo accade restituisce 1.

Independent set Dato un grafo $G(V, E)$ non orientato, un insieme indipendente (independent set) I è un sottoinsieme di V tale che, presi due nodi qualunque $u, v \in I$, $(u, v) \notin E$; cioè I non contiene vertici che sono collegati tra loro da archi in E.

Osserviamo che data una copertura in vertici V', l'insieme complementare $I = V - V'$ è un independent set.

Il problema INDEPENDENT-SET consiste nel chiedersi se, dato un grafo $G(V, E)$ e un $k \in \mathbb{N}$, esiste un insieme indipendente I per G di cardinalità $\geq k$.

Rifacendoci al grafo in figura (8) osserviamo che $\{4, 5, 3\}$ è un independent set per tale grafo. Anche questo problema è un problema NP: infatti, dato un certificato, come ad esempio $\{3, 4, 5\}$, posso verificare in tempo polinomiale la soluzione. Il certificato preso ad esempio garantisce infatti che per $k \leq 3$ il problema ha risposta affermativa, mentre non dice nulla per $k > 3$.

Mostriamo esplicitamente che il problema INDEPENDENT-SET è in NP, precisando meglio l'algoritmo di verifica di una soluzione. Sia y un certificato, quindi un sottoinsieme di V di cardinalità k. L'algoritmo considera tutte le possibili coppie di vertici u e v in y e per ciascuna verifica che nel grafo G non ci sia l'arco (u, v) corrispondente. Se questo è vero per ogni coppia l'algoritmo restituisce 1.

3-SAT Il problema 3-SAT prende in input una formula ϕ in forma normale congiunta (CNF), ossia che sia una congiunzione $\wedge$ di clausole $c_1, ..., c_n$, dove ogni clausola c_i è disgiunzione $\vee$ di (al più) tre letterali, che sono variabili booleane o loro negazioni. Avremo ad esempio

$$\phi = c_1 \wedge c_2 \wedge \ldots \wedge c_n$$

dove ogni c_i è della forma

$$c_i = x_1 \vee (\neg x_2) \vee (\neg x_3)$$

Se esiste un assegnamento alle variabili booleane che rende vera ϕ diremo che ϕ è **soddisfacibile** e il problema 3-SAT restituirà come output 1.

Vediamo un esempio: sia $\phi = c_1 \wedge c_2$, con $c_1 = x_1 \vee \neg x_2 \vee \neg x_3$, $c_2 = \neg x_1 \vee x_2 \vee x_3$. Dato ad esempio l'assegnamento $x_1 = 0$, $x_2 = 0$, $x_3 = 0$, riesco a verificare in tempo polinomiale che tale assegnamento rende ϕ vera e quindi soddisfacibile.

4.2 Problemi di decisione e di ottimo

Per ognuno dei problemi del paragrafo precedente è stato enunciato il problema di decisione, in cui la domanda è "è possibile trovare..". Per ciascuno, tuttavia, è

possibile anche enunciare il relativo problema di ottimo, che mira a trovare la soluzione più generale possibile.

Ad esempio, il problema TSP nella versione di ottimo è così formulato: "qual è il tour di *minimo* costo partendo e tornando in u?". Nel problema VERTEX-COVER invece avremo: "Qual è il *più piccolo* insieme di vertici che posso trovare tale che...", nel problema INDEPENDENT-SET: "Qual è il *più grande* insieme di vertici che costituisce un independent set?".

5 Problemi NP-difficili

Come già osservato per ogni problema è interessante e importante conoscere la difficoltà di *trovare una soluzione*. La classificazione con P ed NP è un primo passo, ma i problemi NP non sono tutti uguali.

Definizione 15. Un problema B è **difficile** in NP se posso risolvere *ogni* problema A in NP tramite una chiamata di procedura a B in tempo polinomiale.

Intuitivamente i problemi NP-difficili sono i più difficili problemi in NP, poiché sono difficili almeno tanto quanto ogni altro problema in NP. Un esempio di problema NP-difficile è il problema VERTEX-COVER.

Una definizione costruttiva di problema NP-difficile è la seguente: per ogni problema in NP A, diciamo che B è NP-difficile se esiste una trasformazione f dell'input w di A in un input $f(w)$ per B in tempo polinomiale e la risposta di B con input $f(w)$ è la stessa di A con input w.

La tecnica con cui ci si riconduce da un problema in NP A ad un altro problema NP B si chiama **riduzione polinomiale**.

Definizione 16. Un problema $\mathcal{A}$ (il cui linguaggio è $L_\mathcal{A}$) si riduce polinomialmente a $\mathcal{B}$ (il cui linguaggio è $L_\mathcal{B}$) se esiste una funzione *polinomiale* f tale che $w \in L_\mathcal{A}$ se e solo se $f(w) \in L_\mathcal{B}$. Si scrive:

$$\mathcal{A} \leq_p \mathcal{B}$$

Il fatto che f debba essere una funzione polinomiale è fondamentale, in quanto in caso contrario non potremmo confrontare $\mathcal{A}$ e $\mathcal{B}$, perché staremmo compromettendo la complessità del problema di partenza.

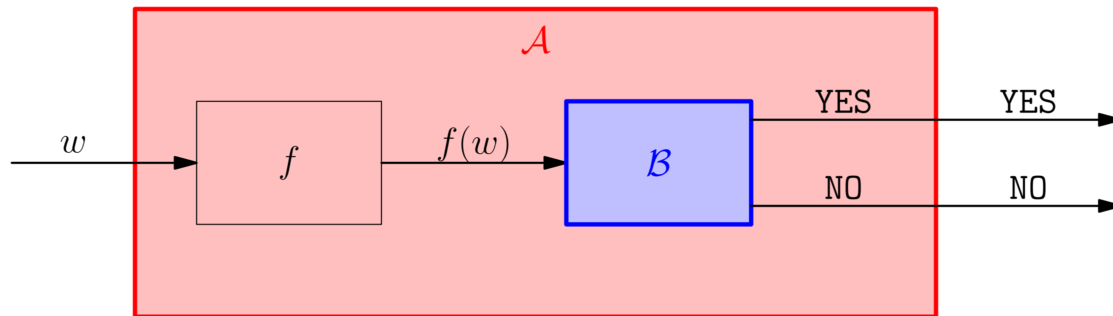


Fig. 9. Riduzione polinomiale da $\mathcal{A}$ (rettangolo rosso) a $\mathcal{B}$ (rettangolo blu): $\mathcal{B}$ risponde sì se anche $\mathcal{A}$ risponde sì, e viceversa. Molto informalmente, è come se $\mathcal{A}$ stesse usando $\mathcal{B}$ come una sua funzione interna.

Definizione 17. Un problema $\mathcal{B}$ è NP-difficile se e solo se **per ogni** $\mathcal{A}$ in NP, $\mathcal{A}$ si riduce a $\mathcal{B}$ in tempo polinomiale, ossia $\mathcal{A} \leq_p \mathcal{B}$.

Definizione 18. Un problema $\mathcal{B}$ si dice NP-completo se è NP-difficile ed è in NP.

Il problema della soddisfacibilità di una formula (SAT) è stato il primo problema dimostrato essere NP-completo (teorema di Cook–Levin). Trovato un problema NP-completo, si può sfruttare questa informazione per dimostrare la NP-completezza di altri problemi. Preso infatti un problema che si sa essere NP-completo, che chiameremo **candidato**, il nostro problema π è NP-completo se e solo se **candidato** $\leq_p \pi$ ed inoltre π sta in NP. Infatti sappiamo che per ogni $A \in$ NP, $A \leq_p$ **candidato** e dunque per transitività $A \leq_p \pi$. Quindi π è difficile almeno quanto A.

Questo è sensato, perché $\forall \mathcal{A} \in$ NP, se esiste f polinomiale che preso un input di $\mathcal{A}$ lo trasforma in un input di **candidato** e g polinomiale che preso un input di **candidato** lo trasforma in un input di π, la funzione che compone f e g sarà ancora una funzione polinomiale e preso l'input del problema $\mathcal{A}$ lo trasformerà nell'input del problema π.

La ragione per cui i problemi NP-difficili sono interessanti è contenuta nel seguente teorema.

Teorema 1. *Se si trovasse un problema π NP-completo in P, allora P=NP.*

Dimostrazione. Per definizione sappiamo che $\forall \mathcal{A} \in$ NP, $\mathcal{A} \leq_p \pi$, cioè esiste una funzione f, che preso un qualsiasi input di $\mathcal{A}$ lo trasforma in un input di π e tale che la risposta restituita dal problema π con input $f(x)$ sia la stessa restituita dal problema $\mathcal{A}$ con input x. Poiché tale f deve inoltre essere polinomiale, e per ipotesi π è risolvibile in tempo polinomiale, segue che anche $\mathcal{A}$ è risolvibile in tempo polinomiale. Poiché il problema $\mathcal{A}$ scelto è arbitrario, allora tutti i problemi in NP sono risolvibili in tempo polinomiale e quindi P=NP. □

5.1 Classici esempi di problemi NP-difficili

I problemi mostrati come classici esempi di problemi NP sono anche NP-difficili.

Independent set

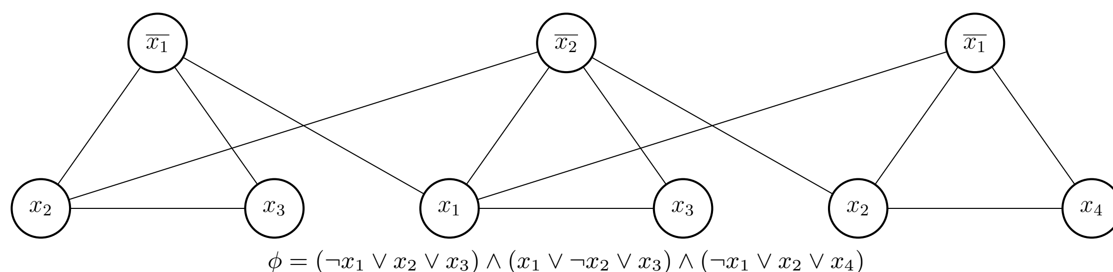
Teorema 2. Independent-Set *è un problema NP-completo*

Dimostrazione. Per dimostrare la proposizione dimostriamo che 3-SAT si riduce a Independent-Set. Per farlo dobbiamo trovare una funzione che presa in input ϕ la trasformi in una coppia (G, k), dove G deve essere un grafo che ha un independent set di dimensione k se e solo se ϕ è soddisfacibile.

Costruiamo tale G nel modo seguente: G contiene 3 vertici per ogni clausola, uno per ogni letterale della clausola; i tre letterali sono collegati da archi (e formano un "triangolo"). Infine, congiungiamo con un arco ciascun letterale al suo negato (qualora presente).

Fissiamo il parametro k uguale al numero di clausole di ϕ.

La funzione di riduzione trasforma quindi una formula ϕ nella coppia (G, k), come descritto sopra. Facilmente si vede che questa trasformazione opera in tempo polinomiale (il numero di vertici di G è lineare rispetto a k e in un qualunque grafo il numero dei possibili archi è al più quadratico rispetto al numero dei vertici).



$$\phi = (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor x_4)$$

Fig. 10. Esempio di una funzione ϕ trasformata in un grafo G. In questo caso $k = 3$.

Mostriamo che questa corrispondenza fra ϕ e (G, k) è una riduzione corretta, nel senso che vogliamo garantisca che l'output di un problema sia 1 se e solo se anche l'output dell'altro è 1. Dimostriamo separatamente i due versi:

- Sia S un insieme indipendente di dimensione k. Poiché in un triangolo tutti i vertici sono fra loro connessi, S non conterrà più di un vertice per ogni triangolo. Considerando i letterali in S come veri, si ottiene un assegnamento di verità consistente (non è possibile che si generi una contraddizione, ovvero che una variabile e la sua negazione siano entrambe vere, poiché questi due letterali non possono essere entrambi in S visto che c'è un arco che li unisce) e tutte le clausole sono soddisfatte (poiché abbiamo preso un vertice per ogni triangolo, che corrisponde a una clausola).
- Dato un assegnamento che rende vera ϕ, per ogni triangolo prendiamo un letterale che risulta vero nell'assegnamento. Tale insieme costituisce un independent set per considerazioni analoghe a quelle già viste ed ha dimensione k per costruzione.

□

Set cover Dato un universo U di n elementi, una collezione S di m sottoinsiemi di U ($S = \{S_1, \ldots, S_m\}$) e una funzione costo $c : S \to \mathbb{Z}^+$, il problema SET-COVER chiede di trovare una collezione C di sottoinsiemi di S di minimo costo che copra tutti gli elementi di U.

Esempio 5. Consideriamo ad esempio $U = \{1, 2, 3, 4, 5\}$ e S così composta:

$$S_1 = \{1, 2, 3\} \quad S_2 = \{1, 4\} \quad S_3 = \{4, 5\} \quad S_4 = \{2, 3\}$$

Ipotizziamo inoltre che la funzione di costo abbia valore 1 per qualunque sottoinsieme S_i.

In questo caso la risposta al problema SET-COVER è $C = \{S_1, S_3\}$. Infatti, la copertura proposta è costituita da due soli sottoinsiemi e non esiste alcuna copertura composta da un solo sottoinsieme (che avrebbe costo 1). Ha senso ragionare in questo modo in quanto ad ogni sottoinsieme è assegnato lo stesso costo, quindi il problema si riduce a trovare la minima sottocollezione di S la cui unione copra U. In generale, non è detto che ci sia un'unica soluzione che realizza il costo minimo.

Un esempio pratico in cui si utilizza il problema SET-COVER è la distribuzione di ripetitori in un'area. L'obiettivo è mettere meno ripetitori possibili facendo in modo che tutta l'area sia coperta.

Fin qui abbiamo considerato SET-COVER come un problema di ottimo. L'analogo problema di decisione (sempre limitandosi al caso in cui i costi sono tutti uguali), è formulato come segue: dato in input un valore j, esiste una sottocollezione C di insiemi di S la cui unione sia U e tale che $|C| \leq j$?

Teorema 3. SET-COVER *è un problema NP-completo.*

Dimostrazione. Dobbiamo dimostrare che la versione di decisione di SET-COVER è in NP ed è NP difficile.

Parte 1 Set cover è in NP. Presa una sottocollezione C è infatti facile verificarne la cardinalità, che deve essere $\leq j$, e che l'unione degli insiemi in C comprenda ogni elemento di U.

Parte 2 Sapendo che VERTEX-COVER è NP difficile, ci basta mostrare che VERTEX-COVER $\leq_p$ SET-COVER. Per mostrarlo, dobbiamo costruire una funzione polinomiale f che mappi ogni istanza di VERTEX-COVER $(G = (V, E), k)$ in un'istanza di SET-COVER C' e tale che (G, k) e $(U, S, C', j) = f(G, k)$ diano le stesse risposte nei rispettivi problemi $\forall (G, j)$.
L'idea chiave è costruire un insieme per ogni vertice che contiene gli archi incidenti a quel vertice. Quindi poniamo l'universo da coprire uguale all'insieme degli archi ($U = E$), numeriamo i vertici da 1 a n e includiamo in S_i gli archi incidenti al vertice i. Il parametro j del problema SET-COVER viene impostato uguale al parametro k di VERTEX-COVER.
Questa funzione di mappatura si esegue in tempo polinomiale. Mostriamo ora che i due problemi messi in relazione nel modo descritto restituiscono effettivamente le stesse risposte.

- Supponiamo che G abbia una copertura C di al più j vertici. Per costruzione, a questa copertura corrisponde un sottoinsieme C' (di uguale cardinalità e quindi al più j) di sottoinsiemi di U. Poiché C è una copertura per G e U è l'insieme degli archi di G, ogni elemento di U deve essere incluso in almeno un sottoinsieme di C', poiché per definizione di copertura ogni arco ha almeno uno dei due vertici in C.
- Viceversa, supponiamo che C' costituisca un set cover di dimensione al più j. Poiché a ciascun insieme è associato un vertice, sia C l'insieme dei vertici corrispondenti agli insiemi in C'. Chiaramente C ha la stessa cardinalità di C' e quindi anche $|C| \leq j$. Asseriamo C è una copertura di G. Infatti, preso un qualsiasi arco $e \in U$ sicuramente C' contiene almeno un insieme che include l'arco e. Tale insieme corrisponde a un nodo che è estremo di e (perché contiene i suoi incidenti), quindi C deve contenere almeno un estremo di e.

□

Cammino hamiltoniano e TSP Dato un grafo $G(V, E)$, esiste un cammino che visiti ogni nodo esattamente una volta? Assumiamo come fatto che il problema del cammino hamiltoniano è un problema di decisione ed è NP-completo. Una banale variante del cammino hamiltoniano è il ciclo hamiltoniano, che ritorna nel vertice di partenza dopo aver visitato tutti i nodi una e una sola volta.

Teorema 4. *Il problema* TSP *(di decisione) è NP-completo.*

Dimostrazione. Dimostriamo che TSP (nella sua variante di decisione) è NP-difficile presentando la riduzione polinomiale Hamiltonian-Cycle $\leq_p$ TSP . Ricordiamo che TSP prende in input un grafo completo e pesato $G(V, E, w)$, dove w è la funzione di peso, e si chiede: "esiste un cammino da u a u che visita ogni nodo di V in G esattamente una volta ed è di costo $\leq k$?".

Considerato il grafo G in input per il ciclo hamiltoniano, costruisco G' per TSP con gli stessi vertici di G e assegnando peso 1 agli archi presenti in G e peso 2 a tutti gli altri archi.

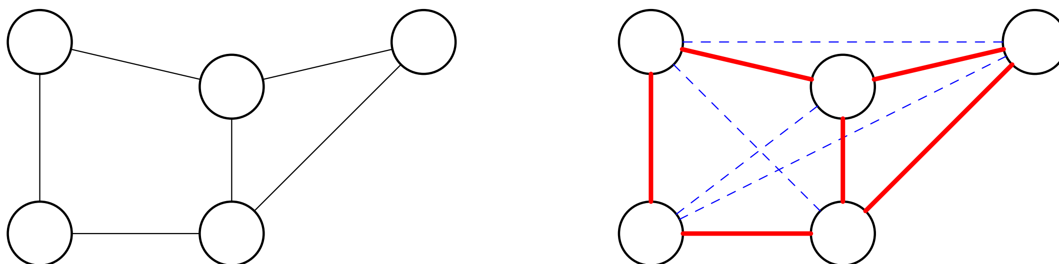


Fig. 11. A sinistra un grafo G (input per Hamiltonian-Cycle). A destra il corrispettivo G' (input per TSP), in cui gli archi originali di G (in rosso) hanno peso 1. I rimanenti (blu tratteggiato) hanno peso 2.

Resta da provare che questa riduzione è ben fondata, ovvero che esiste un ciclo hamiltoniano se e solo se esiste un tour TSP. Dimostriamo separatamente i due versi:

- Posto k uguale al numero di vertici nel grafo G', per costruzione di G' se esiste un cammino di lunghezza $\leq k$ nel problema TSP esso deve per forza essere costituito soltanto da archi di peso 1 (se ci fosse anche solo un arco di peso 2 il costo totale supererebbe k). Tali archi sono anche in G e quindi esiste il ciclo hamiltoniano
- Viceversa, se esiste un ciclo hamiltoniano in G sicuramente è costituito da $|V|$ archi. Percorrendo gli stessi archi in G', otteniamo un cammino per il TSP lungo $|V|$ archi tutti di peso 1, quindi di costo $|V|$. Essendo il limite di costo $k = |V|$, il problema TSP ha risposta affermativa.

6 ϵ-approssimazione

Dato un problema π e un'istanza x, indichiamo con $\mathbf{opt(x)}$ l'ottimo su x di π e con $\mathcal{A}(x)$ il costo di una soluzione ammissibile su x calcolato da un algoritmo $\mathcal{A}$ per il problema π.

Definizione 19. Un algoritmo ϵ-approssimante per un problema π di ottimizzazione è un algoritmo $\mathcal{A}$ *polinomiale* che restituisce una soluzione ammissibile che "dista" da quella ottima al più un fattore costante ϵ. Più precisamente:

- se π è un problema di minimo, allora $\mathcal{A}(x) \leq \epsilon \cdot opt(x)$ con $\epsilon > 1$;
- se π è un problema di massimo, allora $\mathcal{A}(x) \geq \epsilon \cdot opt(x)$, con $0 < \epsilon < 1$.

Osserviamo che tanto più ϵ è vicino a 1, maggiore è l'accuratezza dell'approssimazione. Notiamo inoltre che i vicoli su ϵ nei due casi sono necessari per evitare una contraddizione con l'ottimo (ovvero che la soluzione approssimata venga garantita migliore di quella ottima, assurdo). Il valore ottimo $opt(x)$ è quello che si ottiene eseguendo l'algoritmo esatto per π su input x. Se questo algoritmo esatto ha complessità esponenziale, può essere utile rinunciare all'ottimo per semplificare il problema e ottenere una risposta comunque "buona" in tempo polinomiale. Possiamo allora eseguire in sostituzione dell'algoritmo esatto un algoritmo ϵ-approssimante, che fornisce risposta in tempo polinomiale.

6.1 Un algoritmo 2-approssimante per Vertex-Cover

Costruiamo un algoritmo approssimato per VERTEX-COVER.

Inizializziamo $C = \emptyset$. Dato il grafo $G(V, E)$ in input, consideriamo un arco arbitrario $(u, v) \in E$ e aggiungiamo a C gli estremi u e v. Rimuoviamo quindi da E gli archi coperti da u e v, quindi ripetiamo il procedimento finché E non è vuoto. A questo punto siamo certi di aver ottenuto una copertura; essa ha un numero di vertici pari al doppio degli archi scelti.

Esempio 6. Dato il grafo in figura:

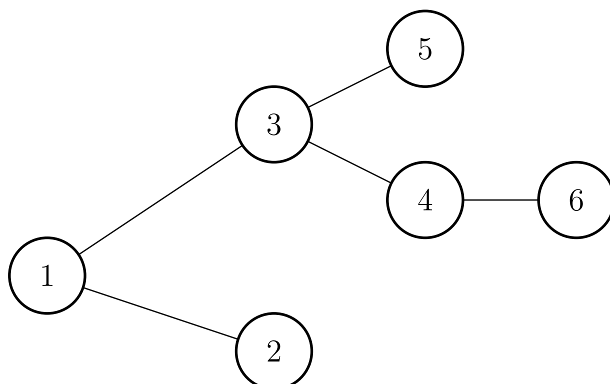


Fig. 12. Input per l'algoritmo approssimato di VERTEX-COVER

L'algoritmo approssimato può ad esempio iniziare a considerare l'arco $(1,3)$: aggiunge 1 e 3 a C ($C = \{1,3\}$) ed elimina gli archi $(1,3)$, $(3,4)$, $(3,5)$, $(1,2)$. L'unico arco rimasto è quindi $(4,6)$, che sarà quello considerato dall'algoritmo all'iterazione successiva. Ora $C = \{1,3,4,6\}$, l'arco $(4,6)$ può essere eliminato e l'algoritmo ha concluso.

Notiamo che la copertura ottenuta ha 4 vertici e non coincide con la soluzione ottima. Infatti, $\{1,3,4\}$ sarebbe stata una copertura di soli 3 vertici, migliore della soluzione trovata dall'algoritmo approssimato.

Chiamiamo **matching** una copertura di *archi* che non condividono estremi. Se un matching ha dimensione k, la dimensione della copertura di *vertici* corrispondente ha dimensione $2k$. Asseriamo che una minima copertura di vertici ha dimensione non inferiore a quella del matching. Infatti, presi due archi e_1 ed e_2 che non condividono vertici, allora ci deve essere almeno un vertice nella copertura per ognuno dei due archi e_1 ed e_2; quindi, dati k archi del matching, la copertura minima V' è tale che $|V'| \geq k$. Detto A l'algoritmo approssimato, sappiamo quindi che $opt(x) \geq k$ e $A(x) \leq 2k$, da cui segue $A(x) \leq 2k \leq 2 \cdot opt(x)$. Abbiamo quindi mostrato che questo algoritmo è 2-approssimante.

6.2 TSP non ha ϵ-approssimazione

Teorema 5. TSP *non ha un'approssimazione costante (a meno che P=NP).*

L'idea è mostrare che se esiste un algoritmo $\mathcal{A}$ ϵ-approssimante per TSP, posso usare $\mathcal{A}$ per risolvere HAMILTONIAN-CYCLE in tempo polinomiale.

Dimostrazione. Sia $G = (V, E)$ un grafo di cui vogliamo determinare se esiste un ciclo hamiltoniano. Costruiamo l'input per TSP, dando un peso particolarmente elevato (e che dipende da ϵ) agli archi non appartenenti E. Formalmente $G' = (V, E', w)$ dove:

$$w((u,v)) = \begin{cases} 1 & (u,v) \in E \\ |V|/(1-1/\epsilon) & \text{altrimenti} \end{cases}$$

Assumiamo che esista un algoritmo $\mathcal{A}$ polinomiale che sia ϵ-approssimante. Possono verificarsi due casi:

1. $\mathcal{A}$ restituisce un tour di costo $|V|$: questo vuol dire che sono stati usati solo archi di peso 1 e quindi esiste il ciclo hamiltoniano.
2. $\mathcal{A}$ restituisce un tour di costo $> |V|$, quindi il tour include almeno uno degli archi costosi (costo totale $\geq |V| - 1 + |V|/(1-1/\epsilon)$). Semplificando si ottiene che $\mathcal{A}(x) > \epsilon|V|$, ma per definizione di ϵ-approssimazione dev'essere $\epsilon \cdot opt(x) \geq \mathcal{A}(x)$ e quindi si ha che $opt(x) \geq A(x)/\epsilon > |V|$, cioè il tour ottimo è di costo strettamente maggiore di $|V|$. Quindi, quando si verifica questo secondo caso sappiamo che il ciclo hamiltoniano non esiste perché non c'è modo di avere un tour di costo $|V|$.

Raggruppando i due casi, $\mathcal{A}$ diventa un algoritmo che decide se esiste o non esiste il ciclo hamiltoniano, che sappiamo essere un problema NP-completo. Poiché abbiamo immaginato $\mathcal{A}$ ϵ-approssimante e quindi polinomiale, seguirebbe che il problema HAMILTONIAN-CYCLE è in P e di conseguenza avremmo dimostrato che P=NP. □

6.3 Un algoritmo 2-approssimante per TSP metrico

Nella sezione precedente abbiamo mostrato che per la versione generale di TSP non esiste alcun algoritmo con approssimazione costante (a meno che P=NP, ritenuto improbabile). Esistono tuttavia varianti di TSP per le quali un tale algoritmo esiste: una di queste varianti è quella in cui la distanza è una *metrica*.

Diciamo che una funzione di costo o distanza $c : V \times V \to \mathbb{R}$ è una **metrica** se:

- i costi sono tutti ≥ 0;
- il costo di un collegamento (u,v) è 0 se e solo se $u = v$;
- comunque presi tre vertici u, v, z vale la disuguaglianza triangolare

$$d(u,v) \leq d(u,z) + d(z,v)$$

In altre parole, la distanza del collegamento diretto tra u e v non supera mai la distanza totale che si ottiene passando per il nodo intermedio z. In uno scenario reale queste ipotesi sono ben fondate e spesso garantite.

Ciclo euleriano Dato un (multi)grafo $G(V,E)$ non orientato, si definisce ciclo euleriano un cammino che inizia in $u \in V$, passa esattamente una volta su ogni $e \in E$ e torna in u.

A differenza del problema del ciclo hamiltoniano, il problema di stabilire se un grafo ammette un ciclo euleriano è "facile" e ammette soluzione in tempo polinomiale. Un teorema facilmente provabile afferma infatti che un grafo ammette un ciclo euleriano se e solo se è connesso e ogni nodo ha grado pari (intuitivamente, perché tutte le volte che si arriva in un nodo si deve poter anche ripartire).

Algoritmo approssimante Possiamo ora descrivere l'algoritmo Double-Tree che dato in input un grafo $G(V, E)$ completo e una funzione di costo c tale che c è una metrica, trova un ciclo hamiltoniano di costo non superiore a due volte il costo di un tour ottimo.

1. Calcola il minimum spanning tree T^* di G (tramite un algoritmo polinomiale).
2. "Raddoppia" ogni arco in T, ottenendo un (multi)grafo euleriano.
3. Trova un ciclo euleriano $\mathcal{E}$ (tramite un algoritmo polinomiale).
4. Trasforma $\mathcal{E}$ in un ciclo hamiltoniano $\mathcal{H}$, partendo da un vertice arbitrario u, procedendo lungo il ciclo e "saltando" i nodi già presenti in $\mathcal{H}$.

Osserviamo che l'ultimo passaggio è lecito solo grazie al fatto che la funzione di costo c è una metrica: questo garantisce che saltando alcuni archi di $\mathcal{E}$ certamente non stiamo percorrendo al loro posto archi di costo maggiore.

Teorema 6. Double-Tree *è un algoritmo 2-approssimante per* TSP *metrico.*

Dimostrazione. Valgono i seguenti fatti:

1. $c(\mathcal{H}) \leq c(\mathcal{E})$: il costo del ciclo hamiltoniano non è superiore al costo del ciclo euleriano, infatti la disuguaglianza triangolare garantisce che l'aver saltato alcuni nodi non ha reso il cammino più costoso.
2. $c(\mathcal{E}) = 2c(T^*)$: il costo del ciclo euleriano è doppio rispetto al costo del minimum spanning tree, visto che è stato ottenuto raddoppiandone gli archi.
3. $c(T^*) \leq c(\mathcal{H}^*)$: il costo del miglior albero ricoprente non è superiore al costo del ciclo hamiltoniano ottimo. Infatti, preso il ciclo hamiltoniano di costo ottimo, rimuovendo un arco otteniamo l'albero (che non può costare di più del ciclo, in quanto i costi degli archi sono non negativi).

Dalla catena di disuguaglianze si ottiene che

$$c(\mathcal{H}) \leq c(\mathcal{E}) = 2c(T^*) \leq 2c(\mathcal{H}^*)$$

e quindi il costo del ciclo hamiltoniano $c(\mathcal{H})$ trovato da Double-Tree non supera due volte il costo del ciclo ottimo $c(\mathcal{H}^*)$.

□

Esiste inoltre l'algoritmo di Christofides, variante migliorativa di quello presentato, che si dimostra essere $3/2$-approssimante.