GRAFI: MINIMO ALBERO RICOPRENTE ALGORITMO DI KRUSKAL

[Deme, seconda edizione] cap. 13

Sezione 13.2

Joseph Kruskal

J. Kruskal was an American mathematician, statistician, computer scientist and psychometrician. He was a student at the University of Chicago and at Princeton University, where he completed his Ph.D. in 1954, nominally under Albert W. Tucker and Roger Lyndon, but de facto under **Paul Erdos** …

He was a Fellow of the American Statistical Association, former president of the Psychometric Society, and former president of the Classification Society of North America. He also initiated and was first president of the Fair Housing Council of South Orange and Maplewood in 1963, and actively supported civil rights in several other organizations.

(continua)

In computer science, his best known work is Kruskal's algorithm for computing the minimal spanning tree (MST) of a weighted graph. The algorithm first orders the edges by weight and then proceeds through the ordered list adding an edge to the partial MST provided that adding the new edge does not create a cycle. Minimal spanning trees have applications to the construction and pricing of communication networks.

In combinatorics, he is known for Kruskal's tree theorem (1960), which is also interesting from a mathematical logic perspective since it can only be proved nonconstructively. Kruskal also applied his work in linguistics, in an experimental lexicostatistical study of Indo-European languages, together with the linguists Isidore Dyen and Paul Black. Their database is still widely-used.

Idea

Un MAR è un **albero**, quindi un **grafo connesso aciclico**, che **contiene tutti i vertici** del grafo di partenza G e con **somma dei pesi degli archi minima**.

Considerando la definizione di MAR e la struttura generica di un **algoritmo greedy** (con appetibilità **non modificabili**), possiamo definire un algoritmo che **costruisce iterativamente un MAR**, a partire da una **foresta vuota** alla quale si aggiungono via via degli **archi** con i seguenti **criteri:**

Appetibilità: l'appetibilità di un **arco è inversamente proporzionale** al suo **peso** (il MAR ha **peso** totale **minimo**).

Criterio di ammissibilità: posso aggiungere un **arco** alla soluzione provvisoria **solo se** questo **non crea cicli** (il MAR è **aciclico**)

Criterio per riconoscere una soluzione: l'albero finora costruito è una **soluzione se** e solo se **è connesso** (il MAR è **connesso**)

Derivazione di Kruskal da Greedy1

Greedy1 **($\{a_1, a_2, \ldots a_n\}$)**

A <- insieme vuoto

ordina gli a_i in ordine

non crescente di appetibilità

for each a_i nell'ordine

if $\{a_i\} \cup A$ è **ammissibile**

aggiungi a_i ad A

Kruskal **(G)**

A <- insieme di archi vuoto

ordina gli archi in una sequenza S in ordine **non decrescente di peso**

for each arco (u,v) in S

if (u,v) $\cup$ A **non ha cicli**

aggiungi (u,v) ad A

Nota: nelle slide relative agli algoritmi Greedy avevamo chiamato S (Soluzione) l'insieme via via costruito. Qui lo chiamiamo A (per Albero) e chiamiamo S la Sequenza ordinata degli archi. Non vi confondete.

Nota2: è un'istanza dell'algoritmo Greedy con **appetibilità fisse**. Posso usare un vettore per S (non lo modifico mai).

Controllo dei cicli con UnionFind

Controllare sempre l'**esistenza di cicli** in (u,v) ∪ A è **fortemente inefficiente**.

Notiamo però che l'algoritmo mantiene una **foresta di alberi** (A), che man mano vengono **fusi dall'aggiunta** di **archi**. Si crea un **ciclo se aggiungiamo un arco tra due nodi che appartengono allo stesso albero**.

IDEA: usiamo una UnionFind per **identificare gli insiemi di vertici** che già **appartengono** ad uno **stesso albero**.

Inizialmente, la UnionFind conterrà degli **insiemi unitari** (a volte chiamati **singoletti**) contenenti i vertici di G.

Ogni volta che sto per aggiungere un arco (u,v), cosidero **find(u) e find(v)**. Se sono **uguali**, **u e v appartengono allo stesso albero** e (u,v) causerebbe un ciclo in A.

Altrimenti, posso **aggiungere (u,v) ad A** e fare la **union** dell'insieme contenente u e quello contenente v

Kruskal con UnionFind

***Kruskal* (G)**

A <- insieme di archi vuoto

ordina gli archi in una sequenza S in ordine non decrescente di peso

crea una UnionFind contenente i vertici di G come insiemi iniziali

for each arco (u,v) in S

if ~~(u,v) ∪ A non ha cicli~~ **find(u) ≠ find(v)**

aggiungi (u,v) ad A

union(u,v)

ATTENZIONE: abbiamo visto che la UnionFind si implementa con degli alberi. Anche l'insieme A è un insieme di Alberi. Tuttavia, questi alberi **sono diversi tra di loro**. L'unica cosa che hanno in comune è che se due nodi appartengono ad uno stesso albero in A, apparterranno allo stesso insieme della UnionFind (e quindi allo stesso albero), ma le strutture saranno diverse.

Semplificazione della UnionFind

Nell'algoritmo visto finora, facciamo sempre la solita sequenza di operazioni sulla UnionFind:

if find(u) ≠ find(v)

…

union(u,v)

(altrimenti non aggiungere (u,v) ad A)

Possiamo creare una nuova union così definita:

Boolean K-union(u,v)
    if find(u) ≠ find(v)
        union(u,v)
        return true
    else return false

Semplificazione con K-union

***Kruskal* (G)**

A <- insieme di archi vuoto

ordina gli archi in una sequenza S in ordine non decrescente di peso

crea una UnionFind contenente i vertici di G come insiemi iniziali

for each arco (u,v) in S

if ~~find(u) ≠ find(v)~~ **K-union(u,v)**

aggiungi (u,v) ad A

~~union(u,v)~~

Ottimizzazione sul numero di archi di A

Ricordando che un albero di n nodi ha **n-1 archi**, possiamo fermare le iterazioni appena A contiene n-1 archi

***Kruskal* (G)**

A <- insieme di archi vuoto

ordina gli archi in una sequenza S in ordine non decrescente di peso

crea una UnionFind contenente i vertici di G come insiemi iniziali

count <- 0

for each arco (u,v) in S

if count = n – 1 return A

if K-union(u,v)

aggiungi (u,v) ad A

count <- count + 1

Complessità

- Creazione di S: **O(n+m)**
- Ordinamento di S: **O(m log m)**
- Creazione della UnionFind: **O(n)**
- Il ciclo fa **m iterazioni** nel caso peggiore
  - E ad ogni iterazione fa massimo **2 find** ed **una union**.

Abbiamo visto che il costo **ammortizzato** delle operazioni sulla UnionFind può essere visto come un **O(1)**

TOT: O(n+m)+O(m log m)+O(n)+O(m*3) = **O(n+m)+O(m log m)**

Ma il grafo è connesso quindi **m > n-1**

O(n+m)+O(m log m) = **O(m log m)**

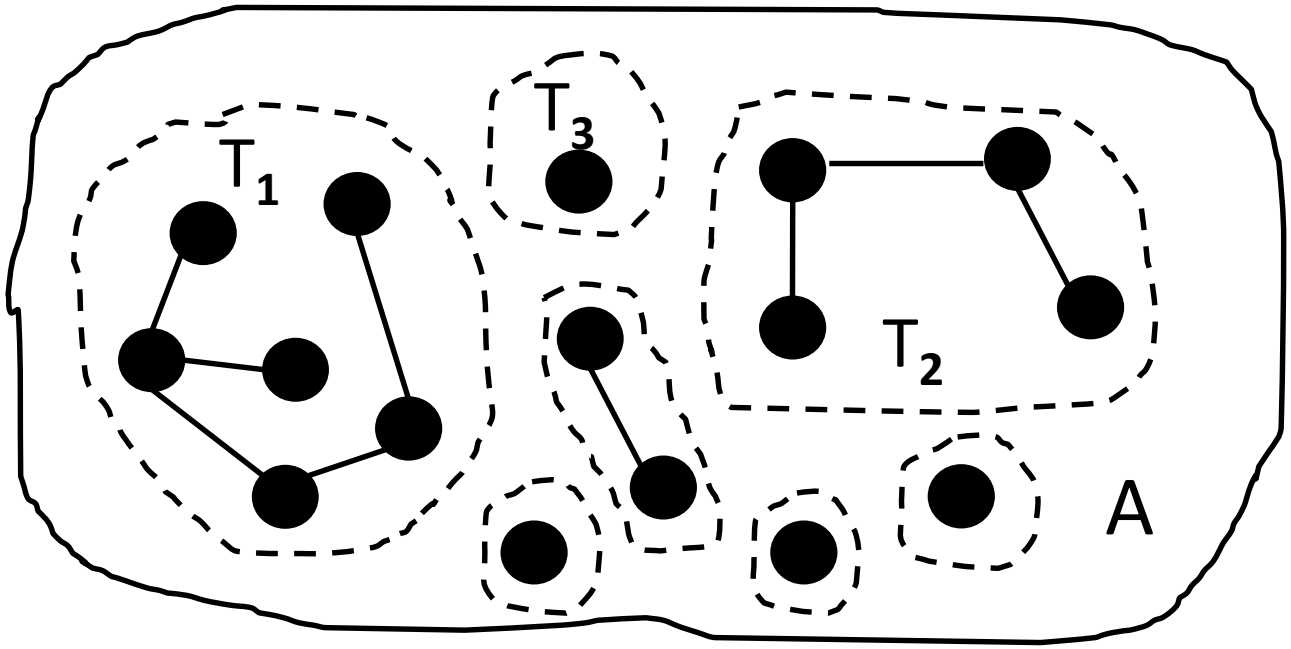
O alternativamente, considerato che **$m = O(n^2)$**

O(m log m) = O(m log n^2) = O(2m log n) = **O(m log n)** **come Prim**

Correttezza

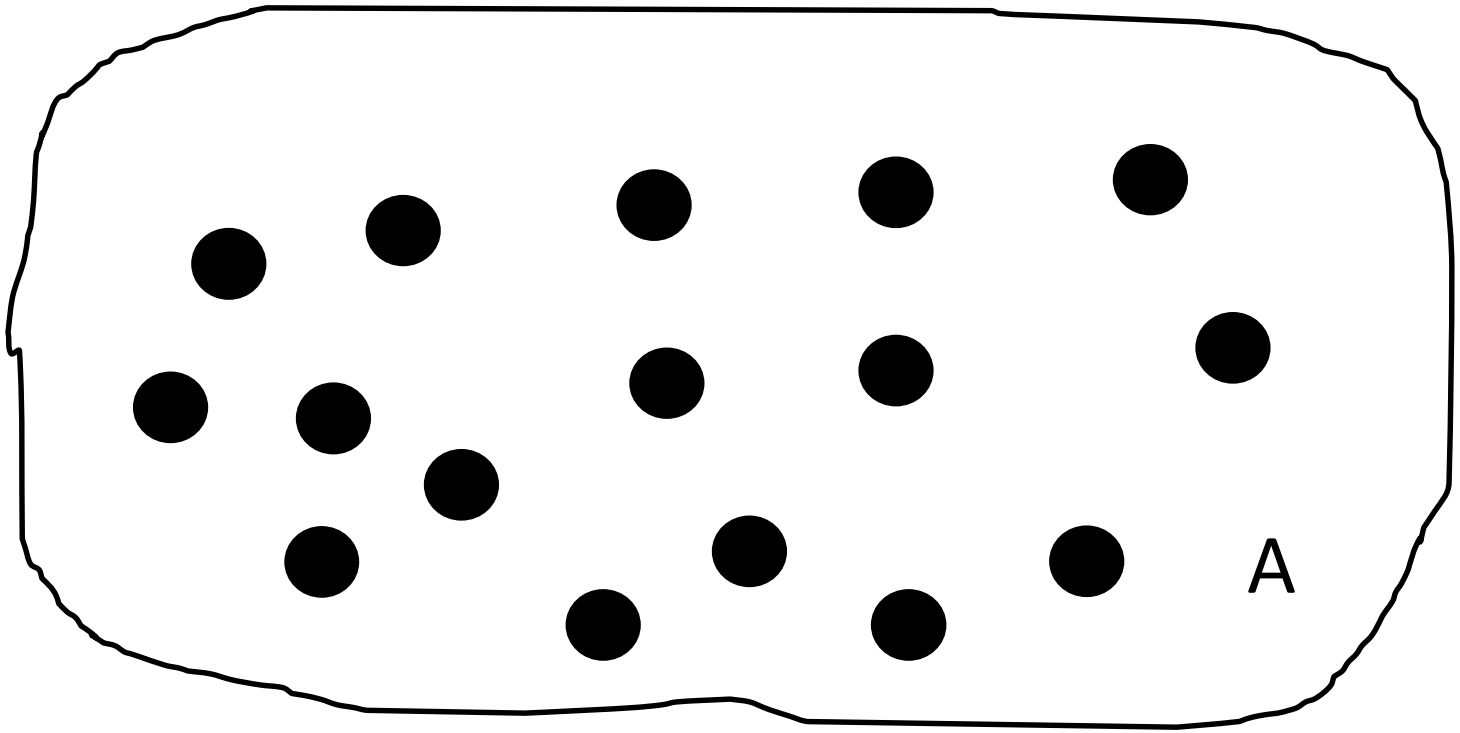
Definiamo un invariante.

Gli archi in **A** definiscono una **foresta** che è un **sottoinsieme di un certo MAR**.



Caso Base

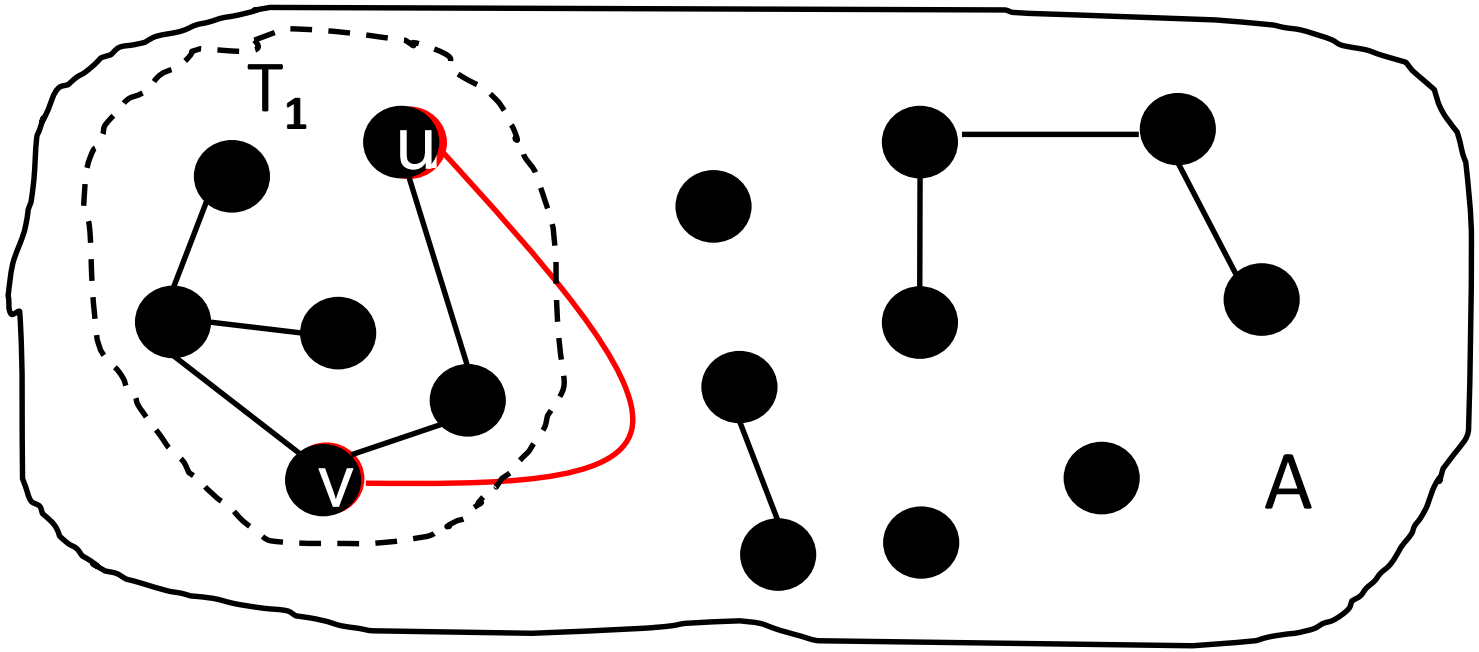
L'invariante è banalmente vero **all'istante iniziale**, quando **A non contiene nessun arco** (come per Prim).



Passo – caso 1

Si sceglie **(u,v)** come **l'arco più leggero non ancora considerato**.

Caso 1: **u e v** appartengono allo **stesso albero**. L'aggiunta di (u,v) ad A creerebbe un **ciclo**, quindi è **impossibile** che **(u,v) ∪ A sia un sottoinsieme di un MAR**. (u,v) viene correttamente **scartato**.

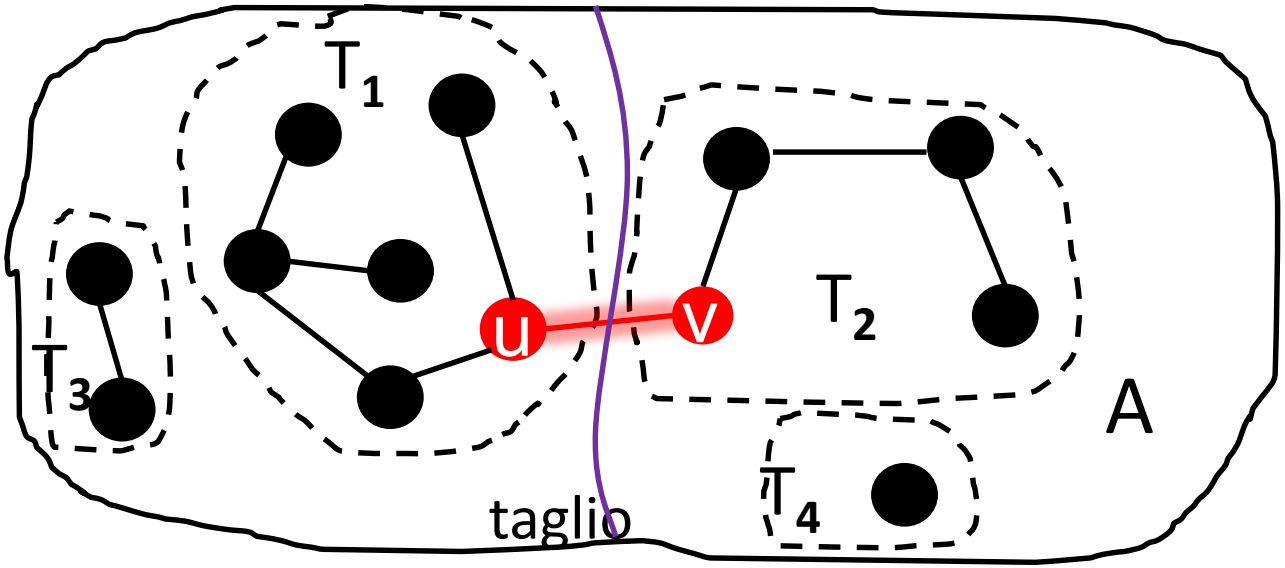


Passo – caso 2

Caso 2: **u e v** appartengono a **due alberi distinti** **T_1 e T_2**

Consideriamo un **taglio** che non tagli **nessun arco di A**, e che abbia **T_1 e T_2** da **parti opposte**, tagliando quindi l'arco (u, v).

L'arco **(u,v)**, essendo quello di peso minimo fra tutti gli archi del grafo che connettono due alberi distinti, è anche **di peso minimo fra tutti gli archi che attraversano il taglio** (dato che questi ultimi sono un sottoinsieme del precedente insieme).



Passo – caso 2

Caso 2: **u e v** appartengono a **due alberi distinti** **T_1 e T_2**

Consideriamo un **taglio** che non tagli **nessun arco di A**, e che abbia **T_1 e T_2** da **parti opposte**, tagliando quindi l'arco (u, v).

L'arco **(u,v)**, essendo quello di peso minimo fra tutti gli archi del grafo che connettono due alberi distinti, è anche **di peso minimo fra tutti gli archi che attraversano il taglio** (che sono un sottoinsieme del preced.).

Quindi, per il **lemma del taglio**, sappiamo che **(u,v) ∪ A è un sottoinsieme di un MAR**.

Quindi aggiungendo (u,v) ad A **manteniamo l'invariante**.

Correttezza - II

Abbiamo quindi dimostrato che **il corpo del ciclo mantiene l'invariante**.

Quindi all'uscita del ciclo:

A è un **sottoinsieme** di un **MAR**

È **connesso**

Contiene **tutti** i **nodi di G**

Al termine dell'esecuzione, **A è un MAR di G**

Cosa devo aver capito fino ad ora

- Derivazione dell'algoritmo di Kruskal dall'algoritmo Greedy 1
- Uso della UnionFind in Kruskal per determinare l'ammissibilità di un arco in una soluzione
- Complessità algoritmo di Kruskal
- Correttezza algoritmo di Kruskal

…se non ho capito qualcosa

- Alzo la mano e chiedo
- Ripasso sul libro
- Chiedo aiuto sul forum
- Chiedo o mando una mail al docente