Anche questo algoritmo risolve il problema del minimo albero ricoprente. In particolare, il minimo albero ricoprente viene costruito mantenendo una foresta alla quale si aggiunge a ogni iterazione un nuovo arco che unisce due sottoalberi distinti. A differenza di Prim, non costruisce l'albero a partire da un nodo scelto come radice, ma come un insieme di archi che alla fine risulta essere un grafo connesso aciclico, quindi un albero libero.

Algoritmo Kruskal

```
Kruskal(G)
S = sequenza archi di G in ordine di costo crescente
T = foresta formata dai nodi di G e nessun arco
While (S non vuota)
    Estrai da S il primo elemento (u,v)
    If (u,v non connessi in T) T = T+(u,v)
Return T
```

$\Rightarrow$ Possiamo ottimizzare siccome un albero di n nodi ha $n-1$ archi quindi possiamo interrompere il ciclo dopo aver raccolto $n-1$ archi. Usiamo una variabile counter. La creiamo sotto a T e la incrementiamo subito dopo la If.

Complessità algoritmo Kruskal

Il problema è controllare se due nodi sono già connessi. Fatto in modo banale con una visita dei due alberi richiede tempo $O(n)$ nel caso peggiore, quindi si ha $O(m \cdot n)$.

Per migliorare l'efficienza possiamo implementare la foresta con una struttura detta Union-Find.

Struttura Union-Find

Servono a rappresentare una collezione di insiemi disgiunti sulla quale siano possibili le seguenti operazioni:

- MakeSet(a): aggiunge l'insieme costituito dal solo elemento a
- Union(A,B): sostituisce gli insiemi A e B con la loro unione
- Find(a): restituisce l'insieme che contiene l'elemento a

Nell'algoritmo di Kruskal, gli insiemi sono le componenti connesse trovate finora, inizialmente costruite con MakeSet; per vedere se due nodi sono già nella stessa componente si usa Find; per fondere due componenti connesse si usa Union.

Se immaginiamo ogni insieme con un suo elemento si ha:

- MakeSet(a) aggiunge l'insieme costituito dal solo elemento a
- Union(A,B): sostituisce gli insiemi rappresentati da a e b con un rappresentante della loro unione
- Find(a) restituisce il rappresentante dell'insieme che contiene l'elemento a

Gli alberi QuickFind permettono di eseguire velocemente la Find. Sono alberi di altezza uno.

- MakeSet(a): aggiunge un nuovo albero con due nodi, radice a e figlio a: $O(1)$
- Union(A,B): rende la radice dell'albero che contiene a padre di tutti i nodi dell'albero che contiene b: $O(n)$
- Find(a): restituisce il padre di a: $O(1)$

Gli alberi Quick-Union permettono di eseguire rapidamente la union. Sono alberi di altezza arbitraria

- makeSet(a): aggiunge un nuovo albero con un unico nodo a: $\boxed{O(1)}$
- union(a, b) tra rappresentanti, rende a padre di b: $\boxed{O(1)}$
- union(a, b) generica, richiede prima una find: $\boxed{O(n)}$
- find(a): risale la catena dei padri di a: $\boxed{O(n)}$

Union by size: la radice dell'albero con meno nodi, diventa figlio dell'albero con cardinalità maggiore

Union by rank: è simile, ma al posto di contare i nodi si usa un campo chiamato rank.

```
Kruskal(G)
S = sequenza archi di G in ordine di costo crescente
T = foresta formata dai nodi di G e nessun arco
UF = struttura union-find vuota
for each (u nodo in G) UF.makeSet(u)
while (S non vuota)
    estrai da S il primo elemento (u,v)
    if (UF.union_by_need(u,v))
        T = T + (u,v)
return T
```

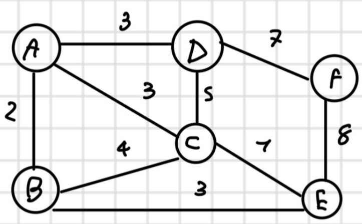
true se UF.find(u) ≠ UF.find(v) (evito un ciclo)

Pseudocodice con struttura union-find

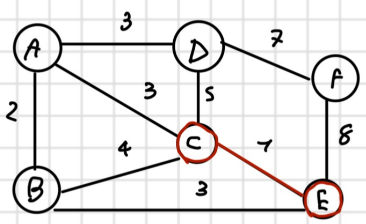
Complessità:
- ordinamento archi: $O(m \log m)$
- makeSet: quasi $O(n+m)$

In totale: $O(m \log m)$

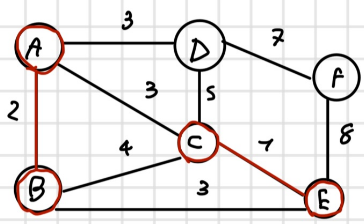
Esempio esecuzione Kruskal



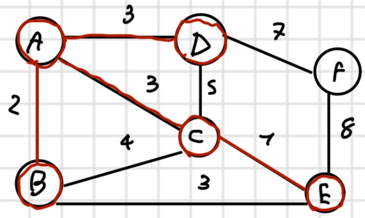
1° passo: selezioniamo l'arco con peso minore, quindi (C, E)



2° passo: si prosegue selezionando l'arco minore senza creare cicli



3° PASSO



NON SCEGLIAMO L'ARCO (B, E) IN QUANTO CI PORTEREBBE A UN CICLO. SCEGLIAMO (D, F)