Algoritmi Greedy

Un **algoritmo Greedy** fa sempre la scelta esatta in un determinato momento, ovvero fa una scelta *localmente ottima* durante una determinata iterazioine nella speranza che tale scelta porterà ad una scelta *globalmente ottima*.

Vengono applicati ai problemi di ottimizzazione.

Gli insiemi delle soluzioni trovate sono dette *ammissibili* e possono essere di due tipi:

- funzione costo
- funzione obbiettivo

La strategia Greedy non può funzionare per ogni problema di ottimizzazione, devo essere sicuro di poterla applicare.
Per avere questa sicurezza, ovvero per avere un algoritmo Greedy funzionante è necessario che valgano due proprietà:

- **Sottostruttura ottima**: un problema ha una sottostruttura ottima se una soluzione ottima del problema contiene al suo interno soluzioni ottime dei sottoproblemi.
- **Proprietà di scelta Greedy**: una soluzione globalmente ottima può essere ottenuta facendo una scelta (greedy) localmente ottima, facciamo la scelta che sembra migliore per il problema corrente senza considerare le soluzioni dei sottoproblemi.

Minimum Spanning Tree (Albero minimo di copertura)

Problema: voglio un albero di copertura che copra ogni nodo e che abbia peso minimo.

Input: $G = (V, E)$ connesso e non diretto, $0 \leq \underbrace{w(u,v)}_{\text{funzione di peso}} \quad \forall (u,v) \in E$

Output: albero di copertura T di costo minimo, ovvero un albero che contenga tutti i nodi nel mio grafo ma dove:

$T = (V, E')$
$E' \subseteq E, E' \text{ deve essere aciclico}$
$|E'| = |V| - 1$

costo: $\sum_{(u,v)\in E} w(u,v) \rightarrow$ la scelta degli archi farà cambiare il costo

Trovare un albero di copertura non è difficile, sommo i pesi degli archi selezionati e trovo il costo ma non è detto che sia il costo minimo.

> Se una parte della soluzione ottima non è ottima allora la soluzione non è ottima

Scelta Greedy 1 (Kruskal)-1956

- ordino gli archi in ordine crescente di peso
- ad ogni iterazione scelgo arco più leggero che non crei cicli

Scelta Greedy 2 (Prim)-1957

Partiamo da una sorgente determinata ($s \in V$)

$E' = \{\}$

$\underbrace{S}_{\text{insieme dei nodi nell'albero}} = \{s\}$

```
while |S| < |V| (fino a quando non ho preso tutti i nodi)
   scelgo l'arco (u, v) di costo minimo tale che u ∈ S
                                          ma ∈/ S
   S := S u {V}
   E' := E' u {(u, v)}
```

Teorema

Sia $\{\} = S \subset U$ un sottoinsieme di V e sia (u, v) un arco di costo minimo tale che $u \in S$ e $v \notin S (v \in V - S)$ allora (u, v) appartiene ad un Minimum Spanning Tree.

Prim

coda con priorità $\rightarrow (\underbrace{v}_{\text{nodo}}, cost(v))$

cost(v) = minimo (v,u) $\in E : u \in S$

S = array da 1 a n

$$S[i] = \begin{cases} 0 & i \notin S \\ 1 & i \in S \end{cases}$$

```
MST_PRIM(G, w)
   S := array[1 ... n]
   for all v ∈ V
      cost[v] := ∞
      prev[v] := NIL
      S[v] := 0

   scelgo s appartenente a V
   cost[s] := 0
   Q := BUILD_QUEUE(V x cost) // tutte le coppie nodo/costo
   while Q is_not_empty
      u := DELETE_MIN(Q)
      S[u] := 1
      for all (u,v) ∈ E
         if S[v] = 0 and cost[v] > w(u, v)
            then
               cost[v] := w(u, v)
               prev[v] := u
               DECREASE_KEY(Q, cost[v])
   Return prev
```

E' molto simile a Dijkstra ma DIVERSO! In questo caso infatti la priorità è il costo di un singolo arco.

Kruskal

Disjoint-Set

E' una struttura dati per gestire gli insiemi disgiunti e mantenere una collezione di sottoinsiemi di elementi.

Disjoint-Set

$S = \{S_1, ..., S_k\}$
$X = \{X_1, ..., X_t\} elementi$
$S_i \subseteq C$

S_i disgiunti $\forall i \neq j, S_i \cap S_j = \{/\}$

$U_{i=1}^{k} S_i = X$

$\forall S_i$ abbiamo un rappresentante R_i

Primitive

- MAKE_SET(x) $\rightarrow$ crea un insieme con x e x è il rappresentante, $x \in X$
- UNION(x, y) $\rightarrow$ unisce insieme che contiene x e insieme che contiene y, $x, y \in X$
- FIND_SET(x) $\rightarrow$ R_x è il rappresentante dell'insieme che contiene x

```
MST_KRUSKAL(G, w)
   for all v ∈ V
      MAKE_SET(v)
   E' := {/}
   ordino archi di E in ordine non decrescente di peso <-- O(mlogn)
   for all (u, v) ∈ E in ordine...                      <-- n * O(FIND_SET
      if FIND_SET(u) != FIND_SET(v)
         then
            E' := E' U {(u, v)}
            UNION(u, v)
   Return E'
```

Make Set

Creo un albero composto da un solo nodo in cui il puntatore al padre punta al nodo stesso.

```
MAKE_SET(DS, x)
   DS.p[x] := x
   DS.rank[x] := 0
```

Find Set

```
FIND_SET(DS, x)
   if DS.p[x] = x
      then return x
      else return FIND_SET(DS, DS.p[x])
```

x\

Union

```
UNION(Ds, x, y)
   Xr := FIND_SET(DS, x)
   Yr := FIND_SET(DS, y)
   if DS.rank[Xr] > DS.rank[Yr]
      then DS.p[Yr] := Xr
      else DS.p[Xr] := Yr
           if DS.rank[Yr] = DS.rank[Xr]
               DS.rank[Yr] := DS.rank[Yr] + 1
```

Albero con radice R ha almeno $2^{rank[R]}$ nodi.

$k \geq 2^{rank[R]}$ $\quad Xr \in S1 \subseteq X$

$logk \geq rank[R]$ $\quad Yr \in S2 \subseteq X$

$rank[Xr] > rank[Yr]$

$$\underbrace{|S1 \cup S2|}_{\text{n° di nodi nell'insieme dei due}} > |S1| \geq 2^{rank[Xr]}$$

$rank[Xr] = rank[Yr]$

$$|S1 \cup S2| = |S1| + |S2| \geq 2^{rank[Xr]} + \geq 2^{rank[Yr]}$$
$$= 2 \cdot 2^{rank[Yr]} = 2^{rank[Yr]+1}$$