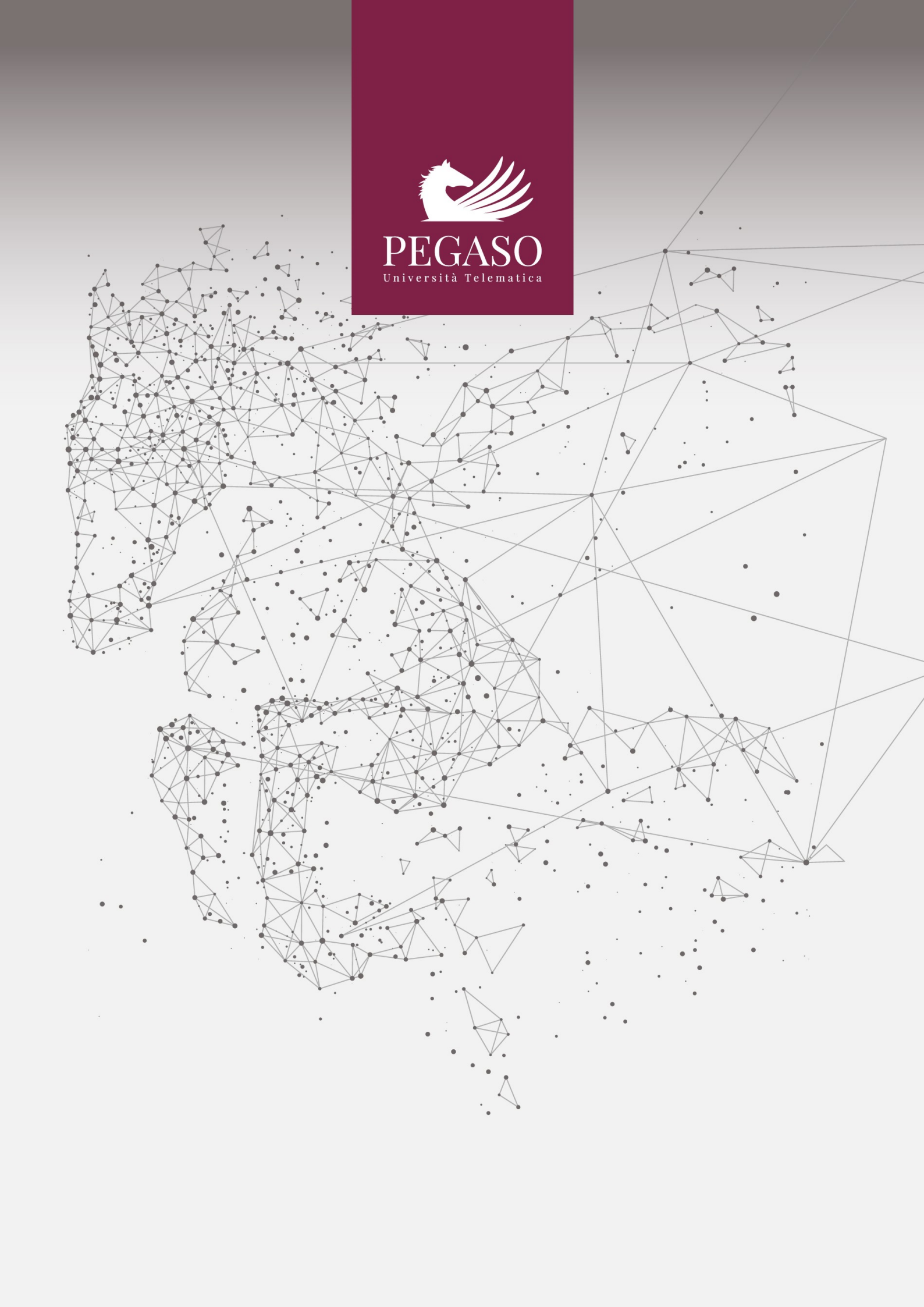
PEGASO
Università Telematica

Indice

1. Algoritmo di Bellman-Ford

Rispetto all'algoritmo di Dijkstra, tale algoritmo può essere utilizzando anche su grafi con archi di peso negativo ma è da un punto di vista computazione, più costoso rispetto a Dijkstra.

Utilizziamo sempre la struttura dell'algoritmo generico:

```
(int[], int[]) shortestPath(Graph G, Node s)
int[] d = new int[1… G.n] // vettore delle distanze
int[] T = new int[1… G.n] // vettore dei padri
boolean[] b = new boolean[1… G.n] // b[u]=true se u ∈ S
foreach u ∈ G.V() – {s} do
     T[u] = nil
     d[u] = +∞
     b[u] = false
     T[s] = nil
     d[s] = 0
     b[s] = true
DataStructure S = DataStructure(); S.add(s)                    // 1
while !S.isEmpty()
     int u = S.extract()                                       // 2
     b[u] = false
     foreach v ∈ G.adj(u)
          if d[u] + G.w(u, v) < d[v]
               if not b[v]
                    S.add(v)                                   // 3
                    b[v] = true
               else
                    // DO SOMETHING                            // 4
          T[v] = u
          d[v] = d[u] + G.w(u, v)
return (T, d)
```

Personalizziamo l'implementazione utilizzando:

1. Una coda di dimensione n → costo $O(1)$

Queue Q = Queue(); Q.enqueue(s)

2. Estrazione del primo elemento della coda → costo $O(1)$

u = Q.dequeue()

3. Inserimento dell'indice v in coda → costo $O(1)$

Q.enqueue(v)

4. L'azione nel caso v sia già presente in S non è necessaria

In questo tipo di algoritmo, se n è il numero di nodi, posso fare al più n "iterazioni": se vi fosse (per qualche motivo) la necessità di farne un'altra, vorrebbe dire che vi è un ciclo e quindi ci sarebbe un problema (e non potrei applicare l'algoritmo). In altre parole: se il mio grafo ha n nodi, il cammino che li attraversa tutti ed n è il massimo che posso considerare: se potessi farne un altro allora vorrebbe dire che vi è un ciclo.

Logica è la seguente:

- Alla prima iterazione, "potenzialmente" potrei modificare la distanza di tutti gli altri nodi (a parte la radice);
- Alla seconda iterazione, la distanza di tutti i nodi -1 (a parte la radice);
- Alla terza iterazione, la distanza di tutti i nodi -2 (a parte la radice);
- […]

Quindi:

- Per $k = 0$, la $0 - esima$ iterazione consiste nell'estrazione del nodo s dalla coda S;
- Per $k > 0$, la $k - esima$ iterazione consiste nell'estrazione di tutti i nodi presenti in S al termine della iterazione $k - 1 - esima$; inoltre al termine della iterazione k, i vettori T e d descrivono i cammini minimi di lunghezza al più k.

2. Esempio

Consideriamo il seguente grafo:

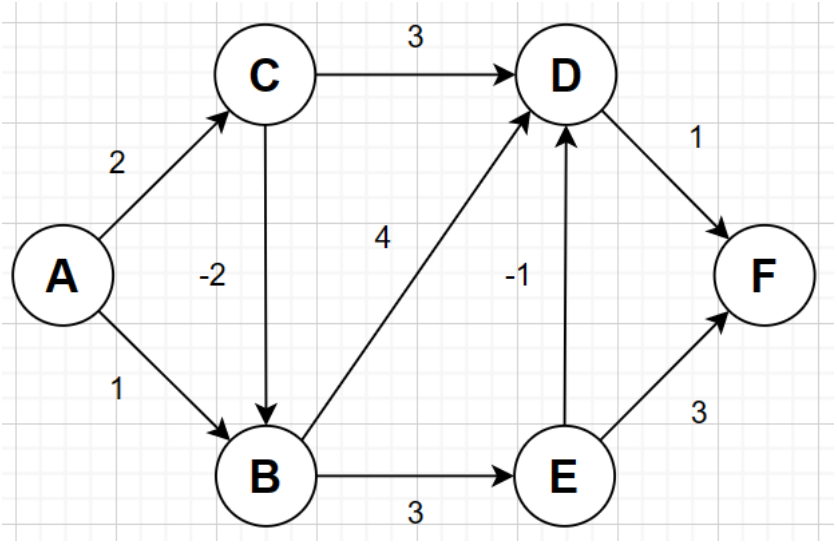

Figura 1: Algoritmo di Bellman-Ford

La situazione iniziale è la seguente:

- $coda = [A]$
- $d = [0, +\infty, +\infty, +\infty, +\infty, +\infty]$
- $T = [nil, nil, nil, nil, nil, nil]$

Alla iterazione 0 occorre estrarre l'elemento dalla coda (in questo caso essendo la iterazione 0 è la radice A):

- $elemento\ estratto = [A]$
  - $d = [0,\ 1,\ 2, +\infty, +\infty, +\infty]$
  - $T = [nil, A, A, nil, nil, nil]$
  - $coda = [B, C]$

Alla iterazione 1 occorre estrarre gli elementi dalla coda (prima B e poi C):

- $elemento\ estratto = [B]$
  - $d = [0,\ 1,\ 2,\ 5,\ 4, +\infty]$
  - $T = [nil, A, A, B, B, nil]$
  - $coda = [C, D, E]$
- $elemento\ estratto = [C]$
  - $d = [0,\ 0,\ 2,\ 5,\ 4, +\infty]$

- $T = [nil, C, A, B, B, nil]$
- $coda = [D, E, B]$

Cosa è accaduto in questa iterazione 1?

Quando abbiamo estratto B abbiamo inserito D ed E nella coda ed abbiamo aggiornato le distanze di D ed E (rispettivamente a 5 e 4) ed il vettore dei genitori, indicando B come genitore per D ed E.

Quando abbiamo estratto C abbiamo come vicino D e B; ma D è già presente in coda, mentre B lo re-inseriamo perché la distanza di B viene "aggiornata" (e migliorata) dal momento che il passaggio per C comporterebbe il percorso (A, C, B) con peso complessivo 0 (+2-2). Ovviamente va aggiornato il vettore dei genitori inserendo C come genitore di B.

Alla iterazione 2 occorre estrarre gli elementi dalla coda (prima D, E e poi B):

- $elemento\ estratto = [D]$
  - $d = [0,\ 0,\ 2,\ 5,\ 4,\ 6]$
  - $T = [nil, C, A, B, B, D]$
  - $coda = [E, B, F]$
- $elemento\ estratto = [E]$
  - $d = [0,\ 0,\ 2,\ 3,\ 4,\ 6]$
  - $T = [nil, C, A, E, B, D]$
  - $coda = [B, F, D]$
- $elemento\ estratto = [B]$
  - $d = [0,\ 0,\ 2,\ 3,\ 3,\ 6]$
  - $T = [nil, C, A, E, B, D]$
  - $coda = [F, D, E]$

Alla iterazione 3 occorre estrarre gli elementi dalla coda (prima F, D e poi E):

- $elemento\ estratto = [F]$
  - $d = [0,\ 0,\ 2,\ 3,\ 3,\ 6]$
  - $T = [nil, C, A, E, B, D]$
  - $coda = [D, E]$
- $elemento\ estratto = [D]$
  - $d = [0,\ 0,\ 2,\ 3,\ 3,\ 4]$
  - $T = [nil, C, A, E, B, D]$
  - $coda = [E]$

- $elemento\ estratto = [E]$
  - $d = [0, 0, 2, 2, 3, 4]$
  - $T = [nil, C, A, E, B, D]$
  - $coda = [D]$

Alla iterazione 4 occorre estrarre l'elemento D dalla coda:

- $elemento\ estratto = [D]$
  - $d = [0, 0, 2, 2, 3, 3]$
  - $T = [nil, C, A, E, B, D]$
  - $coda = [F]$

Alla iterazione 5 occorre estrarre l'elemento F dalla coda:

- $elemento\ estratto = [F]$
  - $d = [0, 0, 2, 2, 3, 3]$
  - $T = [nil, C, A, E, B, D]$
  - $coda = []$

Analizziamo i cammini:

- Per andare in B, il genitore è C; la distanza è pari ad 0
- Per andare in C il genitore è A; la distanza è pari a 2
- Per andare in E il genitore è B; la distanza è pari a 3 → il percorso è (A, C, B, E)
- Per andare in D il genitore è E; la distanza è pari a 2 → il percorso è (A, C, B, E, D)
- Per andare in F il genitore è D; la distanza è pari a 3 → il percorso è (A, C, B, E, D, F)

In termini di complessità avevamo detto che:

1. Queue Q = Queue(); Q.enqueue(s) → $O(1)$

Questa operazione viene svolta 1 sola volta → $O(1)$

2. u = Q.dequeue() → $O(1)$

Questa operazione può essere eseguita (nel caso peggiore) n^2 volte, cioè ad ogni iterazione potenzialmente potrei inserire tutti i nodi nuovamente → $O(n^2)$

3. Q.enqueue(v) → $O(1)$

Questa operazione può essere eseguita (nel caso peggiore) $n \cdot m$ volte, cioè ad ogni estrazione, potenzialmente, devo dover aggiornare tutti gli archi → $O(n \cdot m)$

In totale, possiamo concludere che la complessità dell'algoritmo è $O(n \cdot m)$

Di seguito l'implementazione in Python dell'algoritmo di Bellman Ford:

```
def bellman_ford(graph, start):
    distances = {vertex: float('inf') for vertex in graph}
    distances[start] = 0
    previous_vertices = {vertex: None for vertex in graph}

    for _ in range(len(graph) - 1):
        for vertex in graph:
            for neighbor, weight in graph[vertex]:
               if distances[vertex]+weight< distances[neighbor]:
                 distances[neighbor] = distances[vertex] + weight
                 previous_vertices[neighbor] = vertex

    return distances, previous_vertices
```

3. Esempio pratico: Bellman Ford

Consideriamo il seguente grafo:

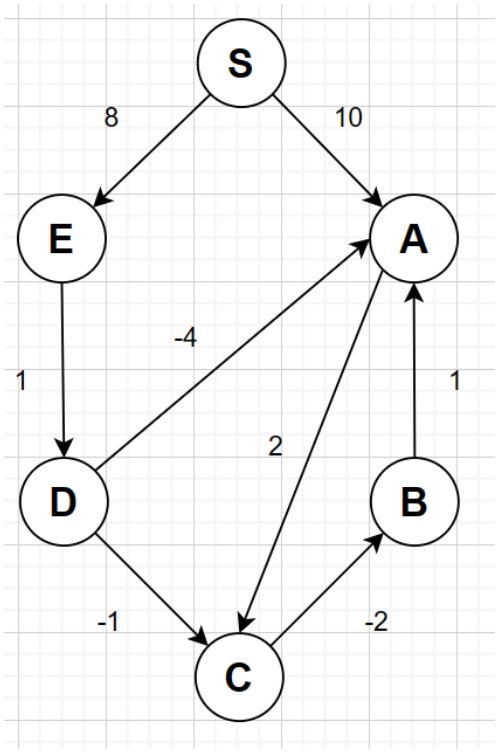

Figura 2: Bellman Ford

Abbiamo 6 nodi, pertanto ci aspettiamo al più 5 iterazioni. Nelle iterazioni, per semplicità esamineremo ogni volta tutti i nodi (per praticità).

Iniziamo con la prima iterazione.

Costruiamo un vettore delle distanze ed uno dei parent:

- $d = [S:0, A:+\infty, B:+\infty, C:+\infty, D:+\infty, E:+\infty]$
- $parent = \{S:nil, A:nil, B:nil, C:nil, D:nil, E:nil\}$

Inizio con il nodo S: da questo posso raggiungere A con un costo 10 ed E con un costo 8:

- $d = [S:0, A:10, B:+\infty, C:+\infty, D:+\infty, E:8]$
- $parent = \{S:nil, A:S, B:nil, C:nil, D:nil, E:S\}$

Proseguo con il nodo A: da questo posso raggiungere C con un costo $d[A] + w(A, C)$=12:

- $d = [S:0, A:10, B:+\infty, C:12, D:+\infty, E:8]$
- $parent = \{S:nil, A:S, B:nil, C:A, D:nil, E:S\}$

Proseguo con il nodo B ma al momento questo non è raggiungibile; quindi, posso andare oltre ed andare al nodo C; da questo posso raggiungere B con un costo $d[C] + w(C,B) = 12 - 2 = 10$:

- $d = [S:0, A:10, B:10, C:12, D:+\infty, E:8]$
- $parent = \{S:nil, A:S, B:C, C:A, D:nil, E:S\}$

Proseguo con il nodo D ma al momento questo non è raggiungibile; quindi, posso andare oltre ed andare al nodo E; da questo posso raggiungere D con un costo $d[E] + w(E,D) = 8 + 1 = 9$:

- $d = [S:0, A:10, B:10, C:12, D:9, E:8]$
- $parent = \{S:nil, A:S, B:C, C:A, D:E, E:S\}$

Proseguiamo con la seconda iterazione.

Riparto da S ma osservando il vettore delle distanze vediamo che non c'è un improvement nel raggiungere i suoi vicini (A ed E) quindi vado avanti con il nodo A.

Anche in questo caso il nodo C è raggiunto con lo stesso valore e quindi nessun improvement: vado avanti con il nodo B.

Anche in questo caso il nodo B è raggiunto con lo stesso valore e quindi nessun improvement: vado avanti con il nodo C.

Anche in questo caso il nodo C è raggiunto con lo stesso valore e quindi nessun improvement: vado avanti con il nodo D.

In questo caso dal nodo D è possibile raggiungere il nodo A con distanza: $d[D] + w(D,A) = 9 - 4 = 5$ e quindi ho migliorato il raggiungimento del nodo A:

- $d = [S:0, A:5, B:10, C:12, D:9, E:8]$
- $parent = \{S:nil, A:D, B:C, C:A, D:E, E:S\}$

Dal nodo D è possibile, inoltre, raggiungere il nodo C con distanza: $d[D] + w(D,C) = 9 - 1 = 8$ e quindi ho migliorato il raggiungimento del nodo C:

- $d = [S:0, A:5, B:10, C:8, D:9, E:8]$
- $parent = \{S:nil, A:D, B:C, C:D, D:E, E:S\}$

Vado avanti con il nodo E ma in questo caso non ci sono improvements.

Proseguiamo con la terza iterazione.

Riparto da S ma osservando il vettore delle distanze vediamo che non c'è un improvement nel raggiungere i suoi vicini (A ed E) quindi vado avanti con il nodo A.

In questo caso dal nodo A è possibile raggiungere il nodo C con distanza: $d[A] + w(A,C) = 5 + 2 = 7$ e quindi ho migliorato il raggiungimento del nodo C:

- $d = [S:0, A:5, B:10, C:7, D:9, E:8]$
- $parent = \{S:nil, A:D, B:C, C:A, D:E, E:S\}$

Vado avanti con il nodo B ma in questo caso non ci sono improvements. Proseguo con il nodo C; dal nodo C è possibile raggiungere il nodo B con distanza: $d[C] + w(C,B) = 7 - 2 = 5$ e quindi ho migliorato il raggiungimento del nodo B:

- $d = [S:0, A:5, B:5, C:7, D:9, E:8]$
- $parent = \{S:nil, A:D, B:C, C:A, D:E, E:S\}$

Vado avanti con il nodo D ma in questo caso non ci sono improvements e stesso discorso per il nodo E.

Proseguiamo con la quarta iterazione e ci rendiamo conto che iterando tra i valori non ci sono improvements: possiamo dunque fermarci a questa iterazione.

Ecco dunque l'albero dei cammini minimi:

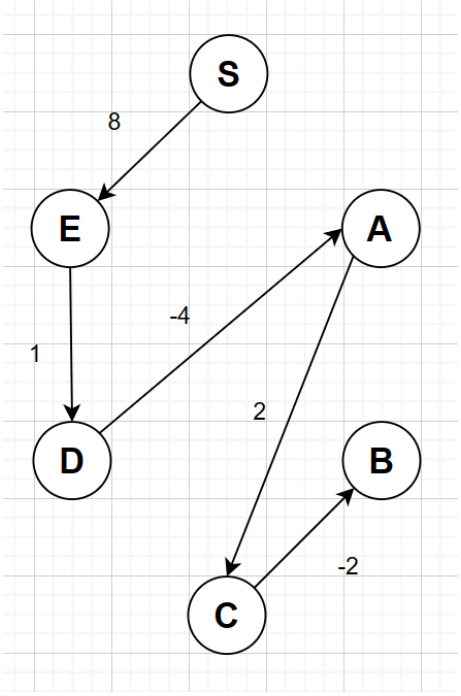

Figura 3: Albero dei cammini minimi - Bellman Ford

4. Cammini minimi in un DAG

In un grafo diretto aciclico, non essendoci cicli, se anche vi fossero pesi negativi, non potrebbero esistere cicli negativi e non posso "ritornare" sullo stesso nodo; pertanto, posso sfruttare questa caratteristica per "rilassare" gli archi in ordine topologico; l'estrazione avviene dunque sfruttando l'ordine topologico.

Di seguito l'algoritmo "semplificato":

```
(int[], int[]) shortestPathDAG(Graph G, Node s)
int[] d = new int[1… G.n] // vettore delle distanze
int[] T = new int[1… G.n] // vettore dei padri
foreach u ∈ G.V()-{s}
     T[u] = nil; d[u] = +∞
     T[s] = nil; d[s] = 0
     Stack S = topological_sort(G)
     while !S.isEmpty()
          u = S.pop()
          foreach v ∈ G.adj(u)
               if d[u] + G.w(u, v) < d[v]
                    T[v] = u
                    d[v] = d[u] + G.w(u, v)
     return (T, d)
```

Riportiamo per comodità l'algoritmo di topological_sort:

```
Stack topological_sort(Graph G)
     Stack S = Stack()
     boolean[] visited = boolean[G.V()]
     foreach u in G.V()
          visited[u] = false
     foreach u in G.V()
          if !visited[u]
               ts_dfs(G, u, visited, S)
     return S
ts_dfs(Graph G, Node u, boolean[] visited, Stack S)
     visited[u] = true
     foreach v in G.adj(u)
          if not visited[v]
               ts_dfs(G, v, visited, S)
     S.push(u)
```

La complessità di questo algoritmo è $O(n+m)$ essendo l'algoritmo topological_sort di complessità $O(n+m)$ ed anche l'operazione di pop ha stessa complessità.

5. Quale algoritmo scegliere

Distinguiamo tra le macro-categorie:

- Pesi positivi:
  - Grafo Diretto Aciclico (DAG) → shortestPathDAG → $O(n+m)$
  - Grafi Densi → Dijkstra → $O(n^2)$
  - Grafi Sparsi → Johnson → $O(m \cdot \log_2 n)$
- Pesi negativi → Bellman Ford → $O(n \cdot m)$

Nel caso in cui non ci sono pesi, possiamo direttamente usare una visita in ampiezza (BFS).

Se indichiamo con $n = numero\ di\ nodi$ ed $m = numero\ di\ archi$, la scelta tra Dijkstra e Johnson si gioca ovviamente sul rapporto tra n ed m.

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione;
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione;
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley;
- Sedgewick: Algoritmi in C, Pearson, 2015;
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione.