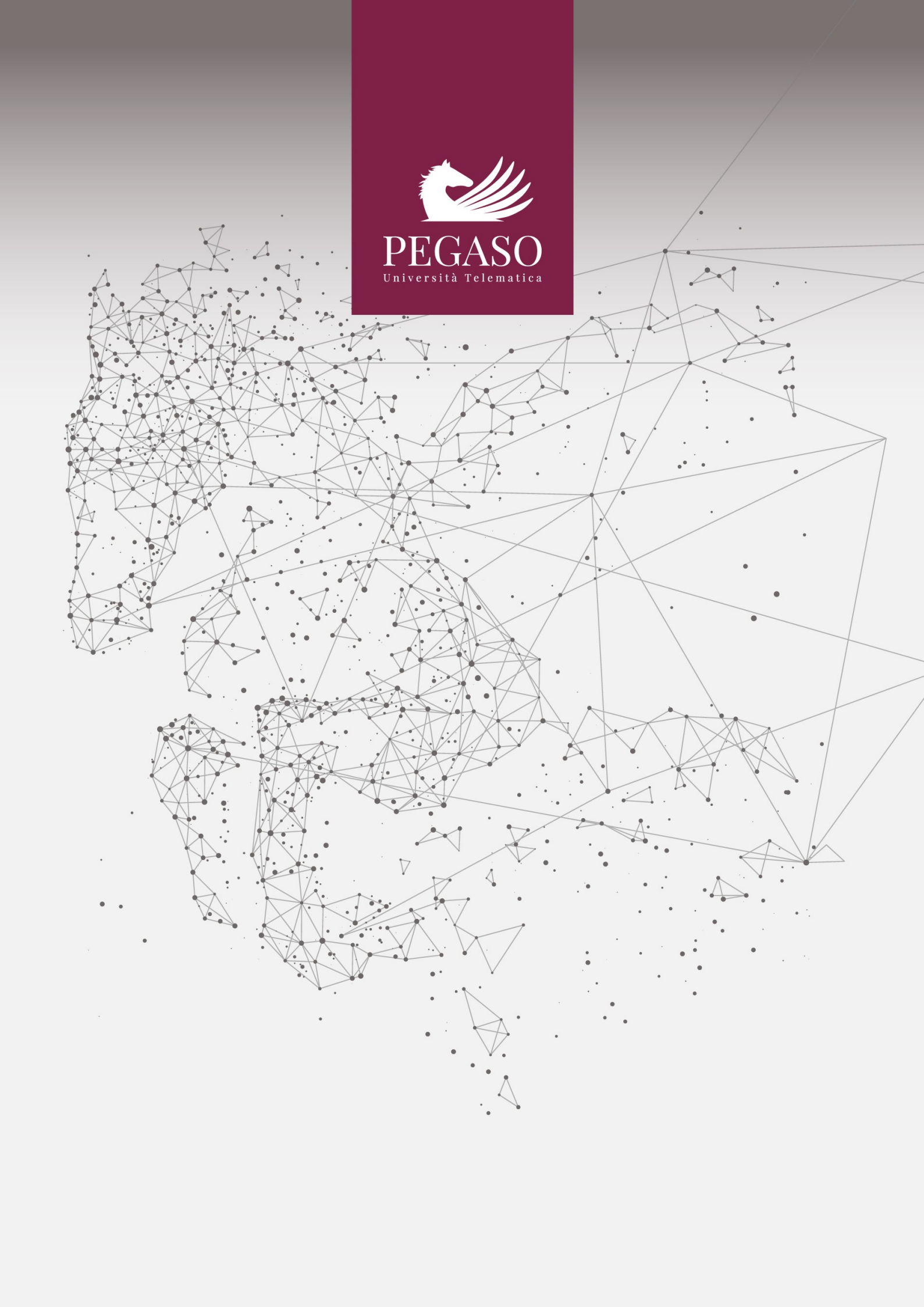

PEGASO
Università Telematica

Indice

1. Algoritmo di Dijkstra

L'algoritmo di Dijkstra è un tipo di implementazione dell'algoritmo "generico" per il calcolo del cammino minimo, e funziona (bene) con pesi positivi.

Tale algoritmo trova applicazione nei protocolli di rete come OSPF.

Riprendiamo l'algoritmo generico ed evidenziamo i punti che necessitano di "customizzazione":

```
(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)
```

Abbiamo perciò 4 punti da "personalizzare":

1. La scelta della struttura dati;
2. La modalità di estrazione dell'elemento dalla struttura dati;
3. L'inserimento del nodo nella struttura dati;
4. L'azione da svolgere nel caso il nodo sia presente nella struttura dati.

Distinguiamo tra 3 versioni dell'algoritmo di Dijkstra:

- Versione base di Dijkstra: coda con priorità basata su vettore;
- Versione di Johnson: coda con priorità basata su heap binario;
- Versione di Fibonacci: heap di Fibonacci.

Partiamo con la versione base di Dijkstra:

- Introduciamo una coda con priorità come struttura dati;
- Si utilizza un vettore di dimensione n per gestire la coda con priorità il cui indice u rappresenta il nodo u-esimo;
- Tutte le priorità vengono inizializzate ad +∞ tranne la priorità di s è posta uguale a 0.

La "personalizzazione" dei 4 punti nell'algoritmo è la seguente:

1. PriorityQueue Q = PriorityQueue(); Q.insert(s, 0);
2. u = Q.deleteMin();
3. Q.insert(v, d[u] + G.w(u, v));
4. Q.decrease(v, d[u] + G.w(u, v)) .

Analizziamo i singoli punti:

1. Inizializziamo una priority queue, cioè un vettore in cui ad ogni nodo è associato un valore di priorità pari ad infinito e per la radice pari a 0 → costo $O(n)$;
2. Finchè la coda non è vuota, si va a cercare dalla coda il minimo e si cancella dalla coda → costo $O(n)$;
3. Si inserisce la priorità del nodo all'interno del vettore → costo $O(1)$;
4. Devo eseguire una operazione di "decrease" della priorità per la mia coda dal momento che ho scoperto un modo migliore per andare ad un particolare modo → costo $O(1)$.

2. Esempio

Consideriamo il seguente esempio:

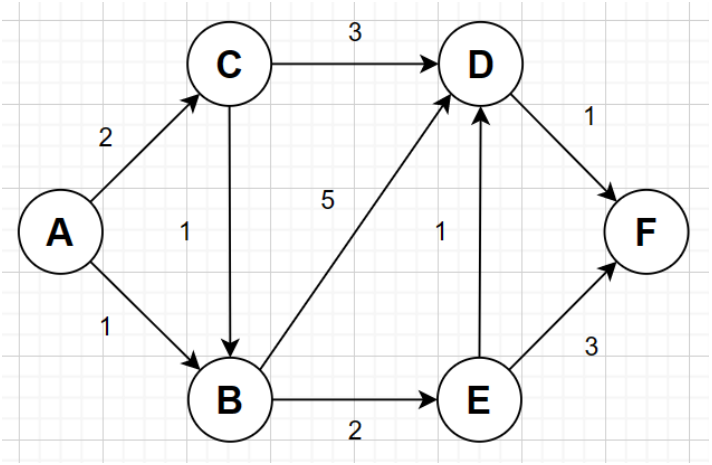


Figura 1: Algoritmo di Dijkstra

Il nodo di partenza è il nodo A, pertanto abbiamo che la coda con priorità è costruita nel seguente modo:

$$coda = [0, +\infty, +\infty, +\infty, +\infty, +\infty]$$

Discorso analogo ma con valori a NIL per il vettore dei padri T:

$$T = [nil, nil, nil, nil, nil, nil]$$

La coda è costituita dai 6 nodi del grafo (da A a F, e che rappresentano dunque gli indici della coda) e tranne per il nodo A che è la sorgente, e quindi la distanza è pari a 0, per tutti gli altri la distanza è pari a $+\infty$. Il vettore dei padri è costituito dai 6 nodi del grafo (da A a F, e che rappresentano dunque gli indici della coda) ed hanno tutti padre posto a NIL.

Ciò che si deve fare inizialmente è estrarre l'elemento della coda con priorità più bassa e quindi si estrae il nodo con priorità 0 che è il nodo A e tale nodo va "cancellato" dalla coda di priorità.

Occorre eseguire la seguente porzione di codice:

```
foreach v ∈ G.adj(u)
     if d[u] + G.w(u, v) < d[v]
          if not b[v]
               Q.insert(v, d[u] + G.w(u, v))
               b[v] = true
          else
               Q.decrease(v, d[u] + G.w(u, v))
     T[v] = u
     d[v] = d[u] + G.w(u, v)
```

Quindi per ogni nodo "adiacente" ad A (cioè B e C), verifico che la condizione sia soddisfatta (ed in tal caso lo è perché le distanze di B e C sono al momento infinite) e quindi vado ad inserire il valore della distanza aggiornato, così come il vettore dei padri

- $coda = [\cancel{0}, 1, 2, +\infty, +\infty, +\infty]$
- $T = [nil, A, A, nil, nil, nil]$

Estraggo l'elemento della coda con priorità più bassa e quindi si estrae il nodo con priorità 1 che è il nodo B e tale nodo va "cancellato" dalla coda di priorità. Quindi per ogni nodo "adiacente" a B (cioè D ed E), verifico che la condizione sia soddisfatta (ed in tal caso lo è perché le distanze di D ed E sono al momento infinite) e quindi vado ad inserire il valore della distanza aggiornato:

- $coda = [\cancel{0}, \cancel{1}, 2, 6, 3, +\infty]$
- $T = [nil, A, A, B, B, nil]$

I valori aggiornati sono derivati dal fatto che:

- $d[D] = d[B] + w(B,D) = 1 + 5 = 6$
- $d[E] = d[B] + w(B,E) = 1 + 2 = 3$

Estraggo l'elemento della coda con priorità più bassa e quindi si estrae il nodo con priorità 2 che è il nodo C e tale nodo va "cancellato" dalla coda di priorità. Quindi per ogni nodo "adiacente" a C (cioè B ed D), verifico che la condizione sia soddisfatta. Ragioniamo sul nodo B, ma in realtà il nodo B era stato cancellato in precedenza, quindi posso "non considerarlo"; analizziamo il nodo D ed in questo caso ho:

- $d[C] + w(C,D) = 2 + 3 = 5 < d[D] = 6$

Quindi posso procedere con l'aggiornamento della distanza per il nodo D:

- $coda = [\cancel{0}, \cancel{1}, \cancel{2}, 5, 3, +\infty]$
- $T = [nil, A, A, C, B, nil]$

Estraggo l'elemento della coda con priorità più bassa e quindi si estrae il nodo con priorità 3 che è il nodo E e tale nodo va "cancellato" dalla coda di priorità. Quindi per ogni nodo "adiacente" a E (cioè D ed F), verifico che la condizione sia soddisfatta.

Analizziamo il nodo D ed in questo caso ho:

- $d[E] + w(E,D) = 3 + 1 = 4 < d[D] = 5$

Quindi posso procedere con l'aggiornamento della distanza per il nodo D:

- $coda = [\cancel{0}, \cancel{1}, \cancel{2}, 4, \cancel{3}, +\infty]$

- $T = [nil, A, A, E, B, nil]$

Analizziamo il nodo F che non è stato ancora visitato pertanto:

- $coda = [\cancel{0}, \cancel{1}, \cancel{2}, 4, \cancel{3}, 6]$
- $T = [nil, A, A, E, B, E]$

Estraggo l'elemento della coda con priorità più bassa e quindi si estrae il nodo con priorità 4 che è il nodo D e tale nodo va "cancellato" dalla coda di priorità. L'unico nodo adiacente è il nodo F:

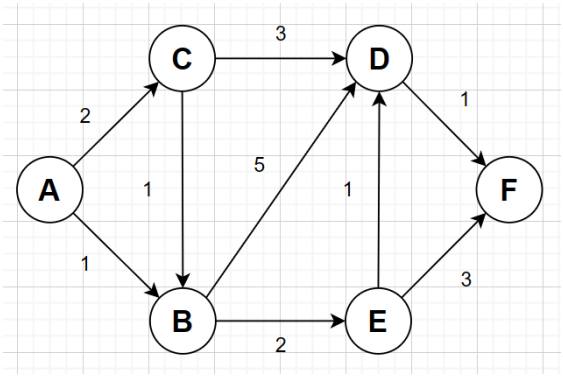
- $d[D] + w(D, F) = 4 + 1 = 5 < d[F] = 6$

Quindi posso procedere con l'aggiornamento della distanza per il nodo F:

- $coda = [\cancel{0}, \cancel{1}, \cancel{2}, \cancel{4}, \cancel{3}, 5]$
- $T = [nil, A, A, E, B, D]$

Termino dunque con l'estrazione del nodo F.

Quindi osservando le due strutture dati posso trovare correttamente i cammini:



Analizziamo i cammini:

- Per andare in B, il genitore è A; la distanza è pari ad 1
- 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, B, E)
- Per andare in D il genitore è E; la distanza è pari a 4 → il percorso è (A, B, E, D)
- Per andare in F il genitore è D; la distanza è pari a 5 → il percorso è (A, B, E, D, F)

In termini di complessità avevamo detto che:

1. PriorityQueue Q = PriorityQueue(); Q.insert(s, 0) → $O(n)$

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

2. u = Q.deleteMin() → $O(n)$

Questa operazione viene svolta n volte → $O(n^2)$

3. Q.insert(v, d[u] + G.w(u, v)) → $O(1)$

Questa operazione viene svolta n volte → $O(n)$

4. Q.decrease(v, d[u] + G.w(u, v)) → $O(1)$

Questa operazione viene svolta m volte → $O(m)$

In totale, possiamo concludere che la complessità dell'algoritmo è $O(n^2)$.

Nella versione di Johnson dell'algoritmo, viene utilizzato un heap binario. In tal caso la valutazione della complessità è la seguente:

1. PriorityQueue Q = PriorityQueue(); Q.insert(s, 0) → $O(n)$

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

2. u = Q.deleteMin() → $O(\log_2 n)$

Questa operazione viene svolta n volte → $O(n \cdot \log_2 n)$

3. Q.insert(v, d[u] + G.w(u, v)) → $O(\log_2 n)$

Questa operazione viene svolta n volte → $O(n \cdot \log_2 n)$

4. Q.decrease(v, d[u] + G.w(u, v)) → $O(\log_2 n)$

Questa operazione viene svolta m volte → $O(m \cdot \log_2 n)$

In totale, possiamo concludere che la complessità dell'algoritmo è $O(m \cdot \log_2 n)$

Nella versione di Fredman-Tarjan dell'algoritmo, viene utilizzato un heap di Fibonacci.

In tal caso la valutazione della complessità è la seguente:

1. PriorityQueue Q = PriorityQueue(); Q.insert(s, 0) → $O(n)$

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

2. u = Q.deleteMin() → $O(\log_2 n)$

Questa operazione viene svolta n volte → $O(n \cdot \log_2 n)$

3. Q.insert(v, d[u] + G.w(u, v)) → $O(\log_2 n)$

Questa operazione viene svolta n volte → $O(n \cdot \log_2 n)$

4. Q.decrease(v, d[u] + G.w(u, v)) → $O(1)$

Questa operazione viene svolta m volte → $O(m)$

In totale, possiamo concludere che la complessità dell'algoritmo è $O(m + n \cdot \log_2 n)$

Di seguito il codice in Python per realizzare l'algoritmo di Dijkstra:

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

    while heap:
        current_distance, current_vertex = heapq.heappop(heap)
        if current_distance > distances[current_vertex]:
            continue
        for neighbor, weight in graph[current_vertex]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_vertices[neighbor] = current_vertex
                heapq.heappush(heap, (distance, neighbor))

    return distances, previous_vertices
```

3. Esempio pratico: Dijkstra

Consideriamo il seguente grafo:

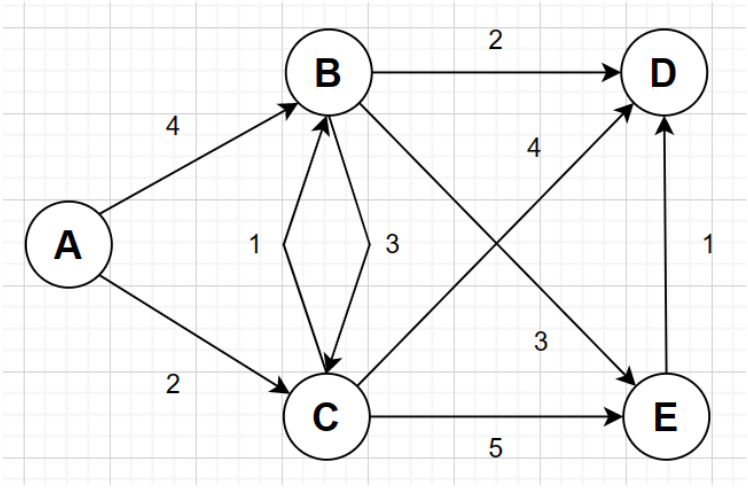


Figura 2: Dijkstra

Applichiamo l'algoritmo di Dijkstra da un punto di vista "pratico".

Utilizziamo 3 strutture:

- $nodi\_non\_visitati = \{A, B, C, D, E\}$
- $distanze = \{A: 0, B: +\infty, C: +\infty, D: +\infty, E: +\infty\}$
- $parent = \{A: nil, B: nil, C: nil, D: nil, E: nil\}$

Iniziamo con l'estrarre il visitare il nodo A; aggiorniamo le distanze dei vicini di A, cioè B e C:

- $d[B] = d[A] + w(A, B) = 0 + 4$
- $d[C] = d[A] + w(A, C) = 0 + 2$
- $nodi\_non\_visitati = \{\cancel{A}, B, C, D, E\}$
- $distanze = \{A: 0, B: 4, C: 2, D: +\infty, E: +\infty\}$
- $parent = \{A: nil, B: A, C: A, D: nil, E: nil\}$

Estraiamo il nodo con distanza minore: C; aggiorniamo le distanze dei vicini di C, cioè B, D ed E:

- $d[B] = d[C] + w(C, B) = 2 + 1 = 3$
- $d[D] = d[C] + w(C, D) = 2 + 4 = 6$
- $d[E] = d[C] + w(C, E) = 2 + 5 = 7$
- $nodi\_non\_visitati = \{\cancel{A}, B, \cancel{C}, D, E\}$

- $distanze = \{A:0, B:3, C:2, D:6, E:7\}$
- $parent = \{A: nil, B:C, C:A, D:C, E:C\}$

Estraiamo il nodo con distanza minore: B; aggiorniamo le distanze dei vicini di B, cioè C, D ed E:

- $d[C] = d[B] + w(B,C) = 3 + 3 = 6$
- $d[D] = d[B] + w(B,D) = 3 + 2 = 5$
- $d[E] = d[B] + w(B,E) = 3 + 3 = 6$
- $nodi\_non\_visitati = \{\cancel{A}, \cancel{B}, \cancel{C}, D, E\}$
- $distanze = \{A:0, B:3, C:2, D:5, E:6\}$
- $parent = \{A: nil, B:C, C:A, D:B, E:B\}$

Estraiamo il nodo con distanza minore: D; tale nodo non ha vicini. Estraiamo infine il nodo E. Ecco, dunque, l'albero dei cammini minimi:

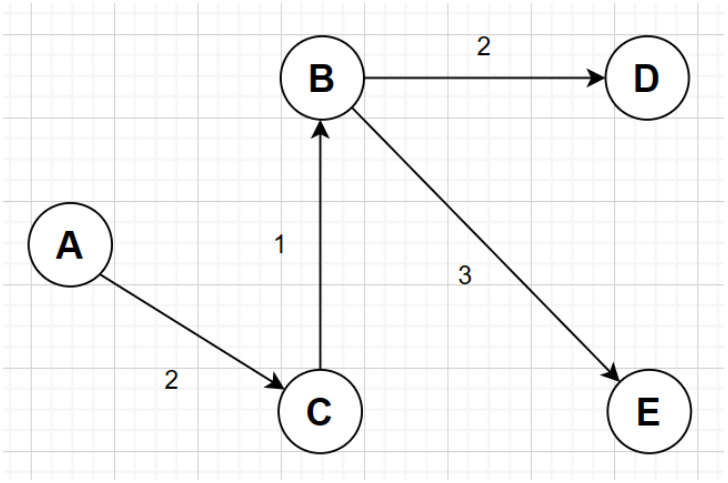


Figura 3: Dijkstra - albero dei cammini minimi

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.