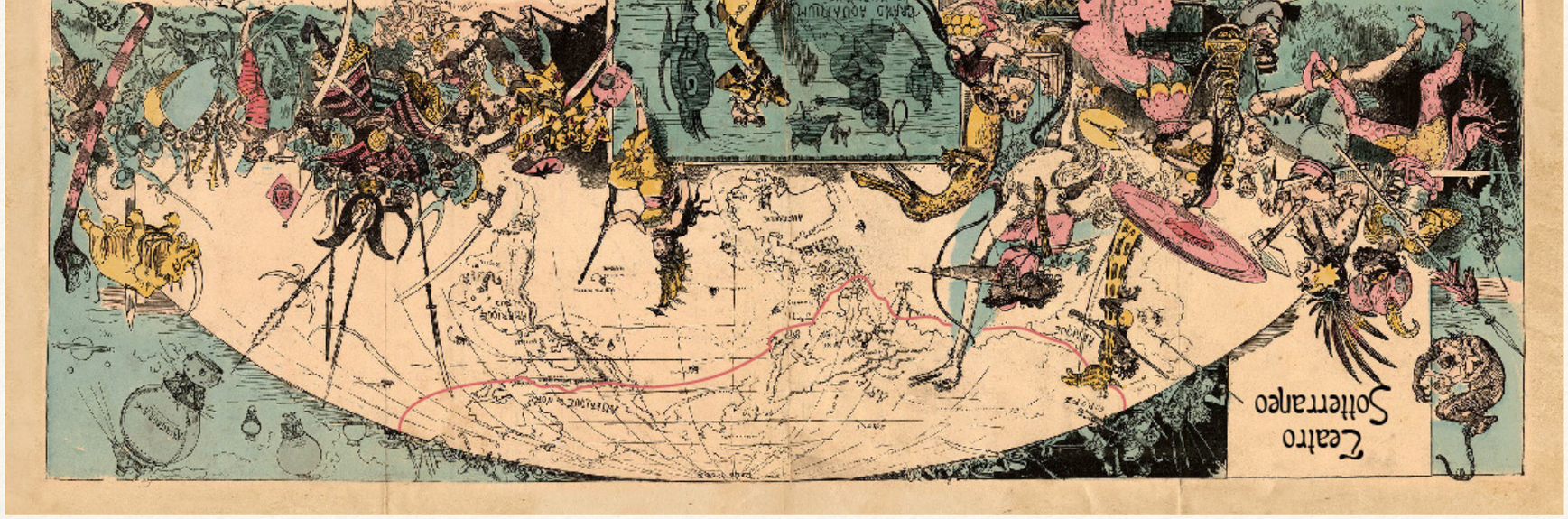
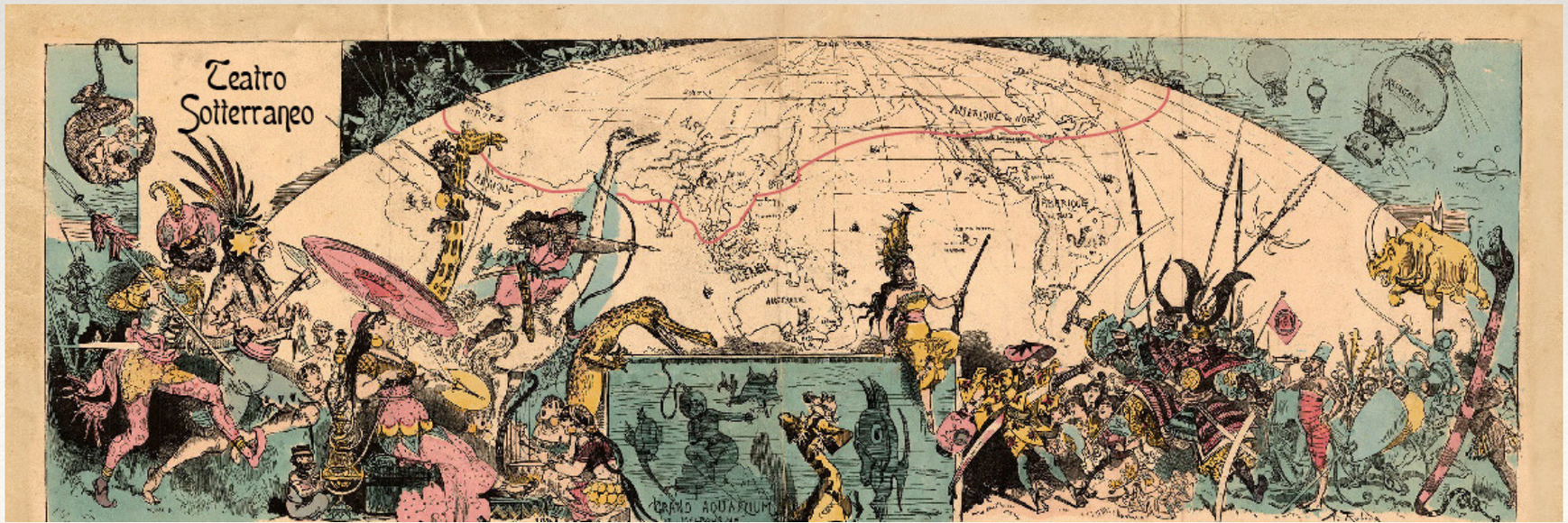


Marco Guzzonato - German Pfaffen - Daniele Cavalli

Presentazione Progetto di Gruppo P8

"Il giro del mondo in 80 giorni?"



1st Step: gestione del database

- Valutando il databases, abbiamo deciso di **estrarre** i seguenti attributi e inserendoli in apposite **liste**

```
for i in range(1,nR):
    nome.append(foglio.cell(i,0).value)
    lat.append(foglio.cell(i,2).value)
    log.append(foglio.cell(i,3).value)
    stato.append(foglio.cell(i,6).value)
    stato2.append(foglio.cell(i,7).value)
    pop.append(foglio.cell(i,9).value)
    iden.append(str(int(foglio.cell(i,10).value)))
```

- Controlliamo la **bontà dei dati** (abbiamo valutato che i dati più sensibili ad errori sono log e lat)
- Poniamo la **popolazione a 0** per dati mancanti ed eliminiamo città con coordinate coincidenti

1st Step: Escamotage

- Per comodità è stata creata una **città fittizia**, con le stesse caratteristiche di quella iniziale, per evitare complicazioni.

```
citta_fine=citta_inizio+"_fine"
fine="1111111111"

for i in range(0,nR):
    if(iden[i]==inizio):
        nome.append(citta_fine)
        lat.append(lat[i])
        log.append(log[i]-0.1)
        stato.append(stato[i])
        stato2.append(stato2[i])
        pop.append(pop[i])
        iden.append(fine)
```

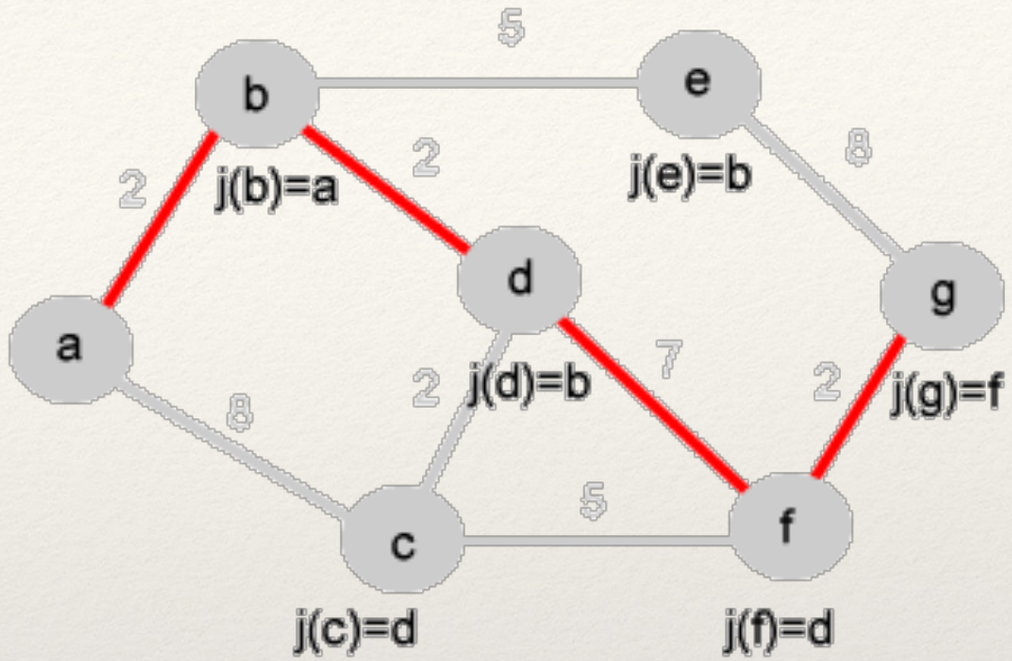
2nd Step: definizione pesi e distanze

- Per definire i pesi, è opportuno conoscere la distanza tra due punti
- Definizione **funzione distanza tra due punti**: in input prende le coordinate e in output restituisce la distanza chilometrica
- Creiamo una **lista** per ogni città, con all'interno le 3 città più vicine
- Creazione di un **dizionario** per ogni città, con dentro il peso delle 3 città più vicine
- I pesi vengono definiti dalla **distanza,** dalla **popolazione** e dallo **Stato**
- I dizionari vengono inseriti in una lista appartenente all'oggetto **nodo**

3rd step: algoritmo

- E usiamo l'**algoritmo di Dijkstra**… a modo nostro.

```
print("Algoritmo di calcolo del grafo utilizzando l'algoritmo Dijkstra")
# funzinamento dell'algoritmo studiato dalla pagina relativa di wikipedia

# imposto la citta di partenza come controllata
for x in nodi:
    if(x.identificativo == inizio):
        x.defi=False
        x.pot=0
        x.coll=inizio
        i=x.ide
k=0
while True:
    for x in nodi[i].adia.keys():
        for y in nodi:
            if (y.identificativo == x and y.defi ):
                somma=nodi[i].pot + nodi[i].adia[x]
                if(y.pot>somma):
                    y.pot=somma
                    y.coll=nodi[i].identificativo
    m=float('inf')
    for z in nodi:
        if(z.defi and z.pot<m):
            m=z.pot
            n=z.ide
    
    nodi[n].defi=False
    i=n
    print(nodi[i].nome)
    # se trova la citta finale  esce e salva il peso come ore di percorso
    if(nodi[i].identificativo == fine ):
        lung=nodi[i].pot
        break
    k=k+1

percorso=[]
percorso_nomi=[]

# creo a ritroso il percoro partendo dalla fine  arrivando alla partenza
s=fine
while True:
    percorso.append(s)
    for x in nodi:
        if(x.identificativo == s):
            s=x.coll
    if(s==inizio):
        percorso.append(s)
        break

# passo da percorso che contine gli identificativi alla stessa lista con i nome relativi delle citta
for x in percorso:
    for y in nodi:
        if(y.identificativo==x):
            percorso_nomi.append(y.nome+" ("+y.stato+")")

# creo una lista per stampare i nomi del percorso
strada=""
for i in range(0,len(percorso_nomi)-1):
    strada=strada+percorso_nomi[len(percorso_nomi)-1-i]+" - "
strada=strada+" "+percorso_nomi[0]

print("--------------------Risultato-----------------------------")
print("Percorso terminato in "+str(lung)+" ore o "+str(float(lung)/24.)+" giorni. (tempo minimo)" )
print("Tragitto: " + strada)
print("----------------------------------------------------------")

```

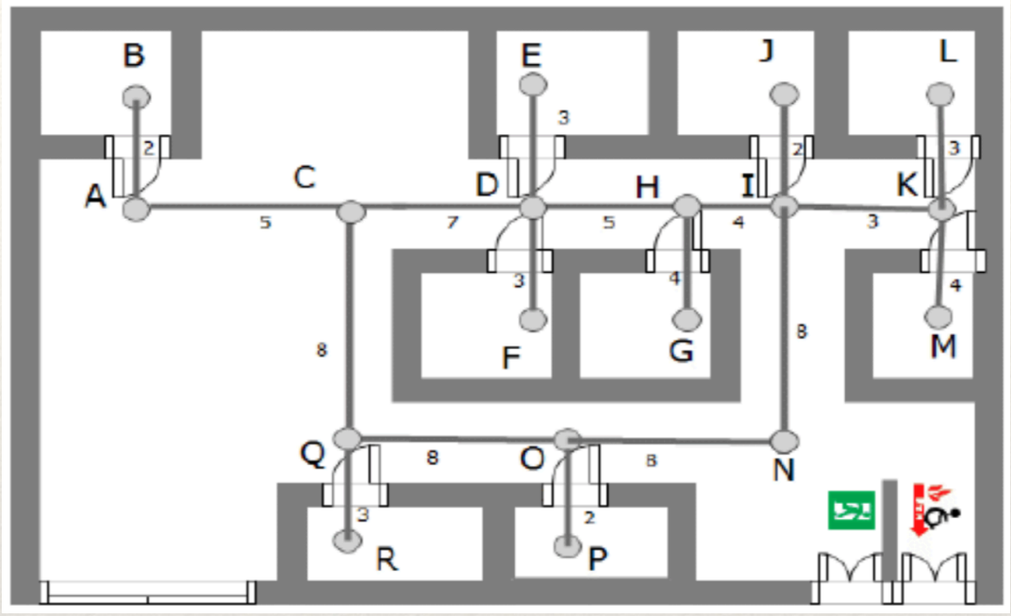


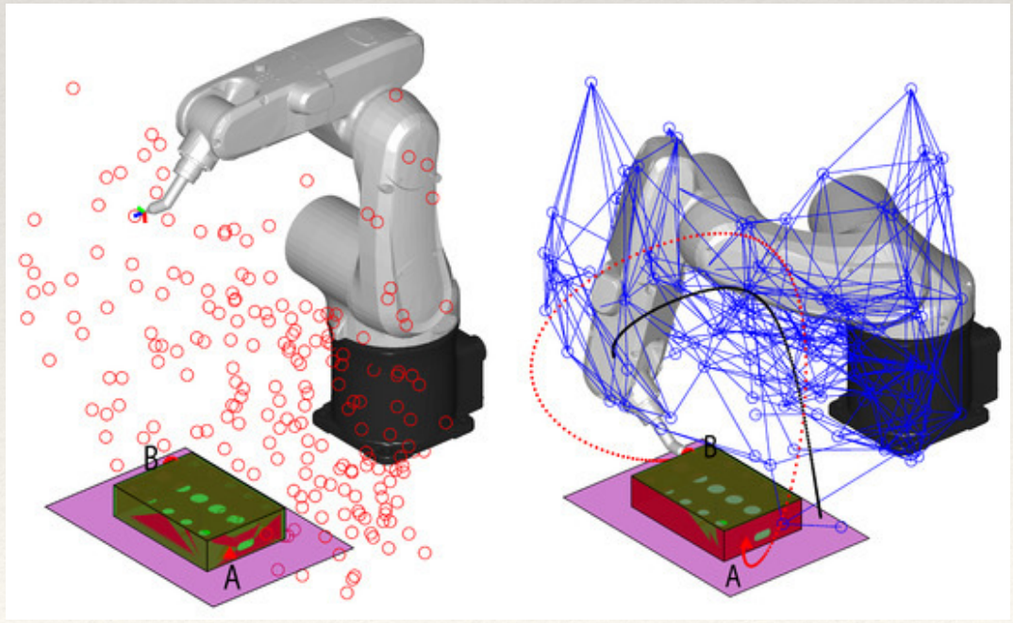
L’algoritmo di Dijkstra

Storia, principi e utilizzo

Storia e Utilizzo

- L'algoritmo di Dijkstra è utilizzato per cercare i **cammini minimi in un grafo** con o senza ordinamento, ciclico e con pesi non negativi sugli archi
- Fu inventato dall'informatico olandese **Edgar Dijkstra** nel 1959.
- Tale algoritmo trova applicazione in contesi quali l'**ottimizzazione nelle realizzazioni di reti** (idriche, telecomunicazioni, circuitali, stradali), ma anche nell'organizzazione e valutazione di percorsi runtime nel campo della **robotica**





L'Algoritmo in breve

- Supponiamo di avere un grafo con **n vertici contraddistinti da numeri interi {1,2,...,n}** e che uno di questi nodi sia quello di partenza e un altro quello di destinazione. Il peso sull'arco che congiunge i nodi **j** e **k** è indicato con ***p(j,k)***. A ogni nodo, al termine dell'analisi, devono essere associate due etichette, ***f(i)*** che indica il **peso totale del cammino** (la somma dei pesi sugli archi percorsi per arrivare al nodo i-esimo) e ***J(i)*** che indica il nodo che precede i nel cammino minimo. Inoltre definiamo due insiemi ***S*** e ***T*** che contengono rispettivamente i nodi a cui sono già state assegnate le etichette e quelli ancora da scandire.

1. Inizializzazione

- Poniamo S={1}, T={2,3,...,n}, f(1)=0, J(1)=0.

- Poniamo f(i)=p(1,i), J(i)=1 per tutti i nodi adiacenti ad 1.

- Poniamo f(i)= ∞, per tutti gli altri nodi.

2. Assegnazione etichetta permanente

- Se f(i)= ∞ per ogni i in T **STOP**
- Troviamo j in T tale che f(j)=min f(i) con i appartenente a T
- Poniamo T=T /{j} e S=S∪{j}
- Se T=∅ **STOP**

3. Assegnazione etichetta provvisoria

- Per ogni i in T, adiacente a j e tale che f(i)>f(j)+p(j,i) poniamo:

  f(i)=f(j)+p(j,i)
  J(i)=j

- Andiamo al passo 2

Pseudo codice

Nel seguente algoritmo, il codice **`u := vertici in Q con la più breve dist[]`**, cerca per dei nodi ***`u`*** nell'insieme dei nodi ***`Q`*** che hanno il valore **`dist[`*`u`*`]`** più piccolo. Questi nodi sono rimossi dall'insieme ***`Q`*** e restituiti all'utente. **`dist_between(`*`u, v`*`)`** calcola la distanza tra due nodi vicini ***`u`*** e ***`v.`*** La variabile ***`alt`*** nelle linee 20 22 rappresenta la lunghezza del percorso dal nodo iniziale al nodo vicino ***`v`*** se passa da ***`u.`*** Se questo percorso è più corto dell'ultimo percorso registrato per *`v,`* allora il percorso corrente è rimpiazzato dal percorso identificato con ***`alt.`*** L'array **`precedente`** è popolato con un puntatore al nodo successivo del grafo sorgente per ricevere il percorso più breve dalla sorgente.

Pseudo codice

```
 1  function Dijkstra(Grafo, sorgente):
 2      For each vertice v in Grafo:                                // Inizializzazione
 3          dist[v] := infinito ;                                   // Distanza iniziale sconosciuta
 4                                                                  // dalla sorgente a v
 5          precedente[v] := non definita ;                              // Nodo precedente in un percorso
ottimale
 6      end for                                                     // dalla sorgente
 7
 8      dist[sorgente] := 0 ;                                         // Distanza dalla sorgente alla sorgente
 9      Q := L'insieme di tutti i nodi nel Grafo ;                        // Tutti i nodi nel grafo sono
10                                                                  // Non ottimizzati e quindi stanno in Q
11      while Q non è vuota:                                       // Loop principale
12          u := vertice in Q con la più breve distanza in dist[] ;    // Nodo iniziale per il primo caso
13          rimuovi u da Q ;
14          if dist[u] = infinito:
15              break ;                                             // tutti i vertici rimanenti sono
16          end if                                                  // inaccessibili dal nodo sorgente
17
18          For each neighbour v di u:                               // dove v non è ancora stato
19                                                                  // rimosso da Q.
20              alt := dist[u] + dist_tra(u, v) ;
21              if alt < dist[v]:                                   // Rilascia (u,v,a)
22                  dist[v] := alt ;
23                  precedente[v] := u ;
24                  decrease-key v in Q;                            // Riordina v nella coda
25              end if
26          end for
27      end while
28  return dist;
```

Pseudo codice

Se siamo interessati solo al percorso minimo tra due nodi ***sorgente*** e ***destinazione,*** possiamo terminare la ricerca alla riga 13 se ***u = destinazione***. Adesso possiamo leggere il percorso più breve da ***sorgente*** a ***destinazione*** tramite un'iterazione inversa:

```
1  S := sequenza vuota
2  u := destinazione
3  while precedente[u] è definito:                    // Costruisci il cammino minimo con uno stack S
4      inserisci u all'inizio di S                  // Esegui il push del vertice sullo stack
5      u := precedente[u]                           // Traverse da destinazione a sorgente.
6  end while ;
```

Adesso la sequenza ***S*** è la lista dei nodi che costituiscono un cammino minimo da ***sorgente*** a ***destinazione***, o la sequenza vuota se non ci sono percorsi minimi esistenti.