Progetto Esame Algorithmic Game Theory

Paola Guarasci — mat 231847

20 febbraio 2022

Descrizione

Scenario iniziale Il problema riguarda la selezione di un sottoinsieme di utenti all'interno di un piu' ampio gruppo di utenti che dovranno effettuare un tour tra alcune citta'. Inoltre bisonga stabilire il costo da addebitare a ciascun utente.

Input L'algoritmo in ingresso si aspetta una lista di tipi, uno per ogni utente. Il tipo rappresenta la località preferita di ogni agente. È stata introdotta una semplificazione riguardo al problema posto originariamente: gli agenti non forniscono una lista di preferenze bensì una sola preferenza di cui ci si può fidare poiché è stato adottato un meccanismo di scelta basato sul mento massimo che ogni agente è disposto a effettuare. Quindi, poiché dell'affermazione sul pagamento ci si può fidare, allora si è deciso di usare questo criterio come discriminante per quanto riguarda l'inserimento del singolo agente nel sottoinsieme travellers, cioè l'insieme dei viaggiatori. In fase di test dell'algoritmo l'input è stato generato casualmente, rispettando i limiti di cardinalità dell'insieme delle località (nessun agente può preferire una località che non esiste).

Grafo Le località sono rappresentate da un grafo pesato non orientato. Ogni località, ovvero ogni nodo, è connessa con tutte le altre località. I pesi sugli archi indicano la distanza in chilometri tra i due nodi alle estremità di ogni arco.

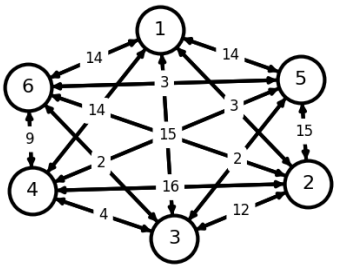

Figura 1: Grafico generale delle localita'

Il calcolo del percorso ottimale e del suo costo è stato modellato usando l'algoritmo di Dijkstra e risolvendo il problema del Commesso Viaggiatore. Per far ciò ci siamo avvalsi di una ottima libreria in Python per la manipolazione di grafi, la libreria NetworkX[1]. La funzione di libreria `approx.greedy_tsp(graph)` prende in input un grafo (o porzione di esso) e ne calcola il cammino hemiltoniano. Per avere il costo del cammino la formula usata è la seguente:

```
cost = sum(graph[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
```

[1]La libreria ha licenza BSD (Open Source) ed e' scaricabile qui https://networkx.org/

Il grafo è modellato come una lista di tuple di archi con tre valori: nodo di partenza, nodo di arrivo, peso. I nodi sono inferiti dalla lista di archi.

Output In uscita si ottiene una lista di viaggiatori e una lista di località selezionate.

Meccanismo Il core del progetto è l'algoritmo di selezione degli agenti. Per ogni agente si valuta la possibilità d'inclusione nel gruppo di viaggiatori, i `travellers`, usando la seguente strategia di selezione:

- Se è stato raggiunto il limite massimo di capienza della macchina, non fare niente.
- Altrimenti, valuta il costo del viaggio tra le località già selezionate e quella nuova preferita dall'agente.
- Se il costo è accettabile e non si superano i vincoli in termini di chilometri massimi, allora inserisci l'agente nel gruppo viaggiatori e la nuova località nel gruppo di località selezionate.
- Infine, per ogni utente già inserito nel gruppo viaggiatori valuta se il nuovo costo del viaggio è accettabile e se non dovesse esserlo elimina l'agente dal gruppo viaggiatori. Elimina anche la località scelta se non è preferita da nessun agente rimasto nel gruppo viaggiatori.

E' un meccanismo diretto la cui modalità di selezione di ogni singolo viaggiatore consente di essere sicuri riguardo l'applicazione di strategie dominanti da parte dei vari agenti, poiché non avviene tramite utilità, che per definizione (vedi traccia) non è affidabile, ma avviene in base al pagamento che ogni agente è disposto a fare (considerato affidabile). Il costo fisso per ogni agente è calcolato dividendo la somma per la cardinalità dell' insieme `travellers`.

Esempio di output Ecco un esempio di output: la prima riga sono gli utenti selezionati, la seconda riga sono le località selezionate, poi costo di viaggio totale e costo di viaggio pro capite.

```
{(5, 34, 21), (5, 72, 20), (3, 88, 18), (6, 7, 20), (2, 52, 16)}
{2, 3, 5, 6}
Num Viaggiatori:  5
Costo viaggio:  45.0
Costo viaggio pro capite:  9.0
```

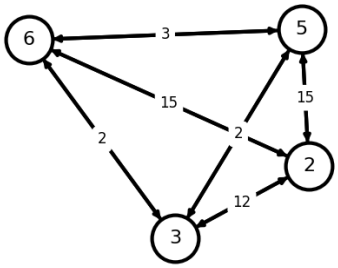

Figura 2: Sottografo con localita' selezionate per il tour

Implementazione

```
"""
  GOAL
  - selezionare gli utenti che eseguiranno il tour
  - deve definire i luoghi che verranno visitati dal tour
  - deve definire i pagamenti che vengono addebitati agli utenti

  VINCOLI
  - A nessun utente deve essere addebitato un costo maggiore rispetto
    a quello che e' disposto a pagare (max)

  NICE TO HAVE
  - il costo del tour, che e' proporzionale alla lunghezza in km,
    deve essere equamente distribuito tra gli utenti del veicolo,
    in modo che nessun agente abbia argomenti per opporsi.
  - il costo fisso deve essere ripartito tra gli agenti in base
    alle utilita' dichiarate, e attuando meccanismi che portino a
    dichiarare in modo veritiero le utilita'.
"""

from importlib.resources import path
from random import randrange
import networkx as nx
import matplotlib.pyplot as plt
from networkx.algorithms import approximation as approx

# https://networkx.org/documentation/latest/reference/algorithms/shortest_paths.
    html

G = nx.DiGraph()

G.add_weighted_edges_from({
    (1, 2, 3),    (1, 3, 17),   (1, 4, 14),   (1, 5, 11),   (1, 6, 8),
    (2, 1, 3),    (2, 3, 12),   (2, 4, 16),   (2, 5, 10),   (2, 6, 6),
    (3, 1, 13),   (3, 2, 12),   (3, 4, 4),    (3, 5, 7),    (3, 6, 5),
    (4, 1, 14),   (4, 2, 15),   (4, 3, 2),    (4, 5, 9),    (4, 6, 3),
    (5, 1, 14),   (5, 2, 15),   (5, 3, 2),    (5, 4, 9),    (5, 6, 3),
    (6, 1, 14),   (6, 2, 15),   (6, 3, 2),    (6, 4, 9),    (6, 5, 3)
})


def printGraph(graph):
  pos = nx.nx_pydot.pydot_layout(G)
  options = {
      "font_size": 16,
      "node_size": 1500,
      "node_color": "white",
      "edgecolors": "black",
      "linewidths": 3,
      "width": 3,
  }
  nx.draw_networkx(graph, pos, **options)
  nx.draw_networkx_edge_labels(graph, pos, edge_labels=nx.get_edge_attributes(
    graph, "weight"), font_size=12, rotate=False)
  ax = plt.gca()
  ax.margins(0.20)
  plt.axis("off")
  plt.show()

def findPath(graph):
    cycle = approx.greedy_tsp(graph, weight="weight")
    try:
      cost = sum(graph[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
    except:
      cost = 0
    return cost

def genUsers(numUsers):
  users = []
```

```
  for i in range(numUsers):
    loc = randrange(1, 7)
    util = randrange(1, 100)
    max = randrange(1, 25)
    users.append((loc, util, max))
  return users

def calcoloCostoFisso():
  pass

users = genUsers(20)
maxUsersInCar = 5
lenMax = 100
fixCost = 10
kmCost = 1.5

location = set()
travellers = set()

for user in users:
  if len(travellers) < maxUsersInCar:
    tmpLocation = location.copy()
    tmpLocation.add(user[0])
    subg = nx.subgraph(G, tmpLocation)
    lenKm = findPath(subg)
    travelCost = float(lenKm * kmCost) / (len(travellers) + 1)
    if (lenKm <= lenMax) and travelCost < user[2]:
      location.add(user[0])
      travellers.add(user)
      for u in users:
        if u[2] < travelCost:
          travellers.discard(u)
          o = [user for user in travellers if user[0] == u[0]]
          if len(o) == 0:
            location.discard(u[0])



print(travellers)
print(location)

totalCost = findPath(nx.subgraph(G, location)) * kmCost

print("Num Viaggiatori: ", len(travellers))
print("Costo viaggio: ", totalCost)
print("Costo viaggio pro capite: ", totalCost / len(travellers))

printGraph(G)
printGraph(nx.subgraph(G, location))
```