Project Exam Algorithmic Game Theory

Paola Guarasci — mat 231847

February 22, 2022

Description

Initial scenario The problem concerns the selection of a subset of users within a larger group of users who will have to tour between certain cities. Furthermore, it is necessary to establish the cost to be charged to each user.

Input The input algorithm expects a list of types, one for each user. The type represents each agent's preferred location. A simplification has been introduced regarding the problem originally posed: agents do not provide a list of preferences but a single preference that can be trusted, since a choice mechanism based on the maximum payment that each agent is willing to make has been adopted. Therefore, since the payment affirmation can be trusted, it was then decided to use this criterion as a discriminant as regards the inclusion of the single agent in the traveller subset, that is, the set of travellers. In the algorithm testing phase, the input was generated randomly, respecting the cardinality limits of the set of locations (no agent can prefer a location that does not exist).

Graph Locations are represented by an undirected weighted graph. Each locality, that is, each node, is connected with all the other localities. The weights on the arches indicate the distance in kilometres between the two nodes at the ends of each arch. The calculation of the optimal

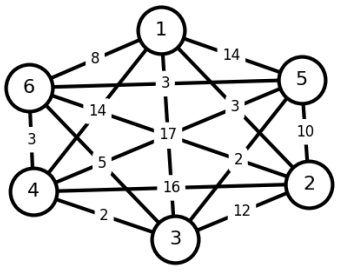

Figure 1: General graph of the localities

route and its cost was modelled using Dijkstra's algorithm and solving the Travelling Salesman problem. To do this, we used an excellent Python library for graph manipulation[1]. The library function `approx.greedy_tsp(graph)` it takes a graph (or portion of it) as input and calculates its Hamiltonian path. To get the cost of the travel, the formula used is the following:

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

The graph is modelled as a list of edge tuples with three values: starting node, ending node, weight. The nodes are inferred from the list of edges.

[1] NetworkX, released under BSD licence (Open Source) https://networkx.org/

Output On output, you get a list of travellers and a list of selected locations.

Mechanism The core of the project is the agent selection algorithm. For each agent, the possibility of inclusion in the group of travellers, the `travellers`, is evaluated using the following selection strategy:

- If the maximum capacity of the machine has been reached, do nothing.
- Otherwise, evaluate the cost of travel between the locations already selected, and the new one preferred by the agent.
- If the cost is acceptable and the constraints in terms of maximum kilometres are not exceeded, then enter the agent in the traveller group and the new location in the selected location group.
- Finally, for each user already included in the traveller's group, it evaluates whether the new cost of the trip is acceptable and if it should not be, it eliminates the agent from the traveller's group. Also delete the chosen location if it is not preferred by any agent remaining in the traveller group.

It is a direct mechanism whose method of selecting each individual traveller allows you to be sure about the application of dominant strategies by the various agents, since it does not take place through utilities, which by definition (see trace) is not reliable, but occurs in based on the payment that each agent is willing to make (considered reliable). The fixed cost for each agent is calculated by dividing the sum by the cardinality of the set `travellers`.

Example of output Here is an example output: the first line is the selected users, the second line is the selected locations, then the total travel cost and pro capite travel cost.

```
$ python src/agt.py
Travellers:  {(2, 98, 13), (5, 93, 24), (4, 51, 23), (2, 10, 21), (1, 84, 24)}
Locations:  {1, 2, 4, 5}
Num Travellers:  5
Travel cost:  54.0
Travel cost for each user:  12.8
```

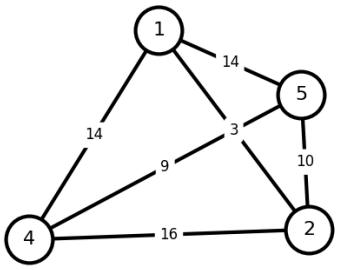

Figure 2: Sub-graph with locations selected for the tour

Implementation

```
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

G = nx.Graph()

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 calculateFixedCost(n, c):
  return c/n;

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

locations = set()
travellers = set()

for user in users:
  if len(travellers) < maxUsersInCar:
    tmpLocation = locations.copy()
```

```
    tmpLocation.add(user[0])
    subg = nx.subgraph(G, tmpLocation)
    lenKm = findPath(subg)
    travelCost = float(lenKm * kmCost) / (len(travellers) + 1) +
    calculateFixedCost(len(travellers) + 1, fixCost)
    if (lenKm <= lenMax) and travelCost < user[2]:
      locations.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:
            locations.discard(u[0])


printGraph(G)

print("Travellers: ", travellers)
print("Locations: ", locations)

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

print("Num Travellers: ", len(travellers))
print("Travel cost: ", totalCost)
print("Travel cost for each user: ", totalCost / len(travellers) +
    calculateFixedCost(len(travellers), fixCost))

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