TdP - ricorsione

Lab 8

Facendo click sul bottone “Worst case analysis” l’applicazione risolve il seguente problema di ottimizzazione mediante un algoritmo ricorsivo:
1. selezionare il sottoinsieme di eventi di blackout (tabella ‘PowerOutages’) che si sono verificati in un massimo di **X** anni, per un totale di **Y** ore di disservizio massimo, tale da massimizzare il numero totale di persone coinvolte. In particolare, l’applicazione deve rispettare i seguenti vincoli:

- Quando si aggiunge un evento di blackout alla lista di eventi selezionati, bisogna considerare tutte le ore di disservizio relative all’evento considerato. Il numero di ore di disservizio viene calcolato come la differenza tra *date_event_began* e *date_event_finished*
- Il numero totale di ore di disservizio del sottoinsieme di eventi selezionati deve essere sempre minore o uguale del valore **Y** inserito dall’utente nell’interfaccia grafica
- La differenza tra l’anno dell’evento più recente e l’anno di quello più vecchio deve essere sempre minore o uguale del numero di anni **X** inserito dall’utente nell’interfaccia grafica.

```
def worstCase(self, nerc, maxY, maxH):
      self.loadEvents(nerc)

      parziale = []

      self.ricorsione(parziale, maxY, maxH, 0)


def ricorsione(self, parziale, maxY, maxH, pos):

         # terminazione
         if self.sumDurata(parziale)/60/60 > maxH:
            return

         # verifica se best
         if self.countCustomers(parziale) > self._clientiMaxBest:
             self._solBest = parziale[:]
             self._clientiMaxBest = self.countCustomers(parziale)

         # ricorsione
         i = pos
         for e in self._listEvents[pos:]:
            parziale.append(e)
            if self.getRangeAnni(parziale) > maxY:
                 parziale.remove(e)
                 return
             i+=1
             self.ricorsione(parziale, maxY, maxH, i)
             parziale.remove(e)

def getRangeAnni(self, listOutages):
      if len(listOutages) < 2:
            return 0
      first = listOutages[0].date_event_began
      last = listOutages[-1].date_event_finished
      return int(last.year - first.year)
def countCustomers(self, listOutages):
      if len(listOutages) == 0:
```

```
            return 0
      numCustomers = 0
      for event in listOutages:
            numCustomers += event.customers_affected
      return numCustomers

def sumDurata(self, listOutages):
      if len(listOutages)==0:
            return 0
      sum = 0
      for event in listOutages:
            sum += self.durata(event)
      return sum

def loadEvents(self, nerc):
      self._listEvents = DAO.getAllEvents(nerc)

def loadNerc(self):
      self._listNerc = DAO.getAllNerc()
```

Metro-paris

```
def getBFSNodes(self, source):
      edges = nx.bfs_edges(self._grafo, source)
      visited = []
      for u,v in edges:
            visited.append(v)
      return visited
```

BFS (Breadth-First Search)
Quando usare BFS: -> Preferibile usare questo per percorsi brevi

- **Esplorazione in ampiezza:** BFS è utile quando è necessario esplorare tutti i nodi di un grafo in ampiezza, cioè visitando tutti i vicini del nodo corrente prima di spostarsi ai vicini dei vicini.
- **Trova il percorso più breve:** BFS è ideale per trovare il percorso più breve (in termini di numero di archi) tra due nodi nel caso in cui tutti gli archi abbiano lo stesso peso (non pesati).

```
def getDFSNodes(self, source):
      edges = nx.dfs_edges(self._grafo, source)
      visited = []
      for u,v in edges:
            visited.append(v)
      return visited
```

DFS (Depth-First Search)
Quando usare DFS:

- **Esplorazione in profondità:** DFS è utile quando è necessario esplorare un ramo del grafo fino a quando non si raggiunge il fondo prima di tornare indietro.
- **Ricerca in profondità:** DFS può essere utilizzato per trovare soluzioni in spazi di ricerca, come nei problemi di labirinto o nei problemi di percorso più profondo.
- **Ordinamento topologico:** DFS può essere utilizzato per ordinare i nodi in ordine topologico in un grafo diretto aciclico (DAG).

```
def getBestPath(self, v0, v1):
      costoTot, path = nx.single_source_dijkstra(self._grafo,v0,v1)
      return costoTot, path
```

La funzione `getBestPath` è progettata per calcolare il percorso più breve tra due nodi in un grafo, utilizzando l'algoritmo di Dijkstra. La funzione getBestPath è utile quando hai un grafo e hai bisogno di trovare il percorso più breve tra due nodi specifici.

ArtsMia

```
def getBestPaht(self, lun, v0):
      self._solBest = []
      self._costBest = 0

      parziale = [v0]

      for v in self._grafo.neighbors(v0):
            if v.classification == v0.classification:
                  parziale.append(v)
                  self.ricorsione(parziale, lun)
                  parziale.pop()

      return self._solBest, self._costBest

def ricorsione(self, parziale, lun):

# Controllo se parziale è una sol valida, ed in caso se è migliore del best
      if len(parziale) == lun:
            if self.peso(parziale) > self._costBest:
                  self._costBest = self.peso(parziale)
                  self._solBest = copy.deepcopy((parziale))
             return

# Se arrivo qui, allora len(parziale) < lun
      for v in self._grafo.neighbors(parziale[-1]):
      #v lo aggiungo se non è già in parziale e se ha stessa classif di v0
            if v.classification == parziale[-1].classification and v not in
parziale:
                  parziale.append(v)
                  self.ricorsione(parziale, lun)
                  parziale.pop()

def peso(self, listObject):
      p = 0
      for i in range(0, len(listObject)-1):
            p += self._grafo[listObject[i]][listObject[i+1]["weight"]
      return p
```

```
def getConnessa(self, v0int):
      v0 = self._idMap[v0int]

      #Modo1: successori di v0 in DFS
      successors = nx.dfs_successors(self._grafo, v0)
      allSucc = []
      for v in successors.values():
            allSucc.extend(v)

      print(f"Metodo 1 (pred): {len(allSucc)}")

      #Modo2: predecessori di v0 in DFS
      predecessors = nx.dfs_predecessors(self._grafo, v0)
      print(f"Metodo 2 (succ): {len(predecessors.values())}")

      #Modo3: conto i nodi dell'albero di visita
      tree = nx.dfs_tree(self._grafo, v0)
      print(f"Metodo 3 (tree): {len(tree.nodes)}")

      #Modo4: node_connected_component
      connComp = nx.node_connected_component(self._grafo, v0)
      print(f"Metodo 4 (connected comp): {len(connComp)}")

      return len(connComp)
```

- Metodi 1 e 2 (Successori e Predecessori in DFS):
  - Questi metodi forniscono informazioni parziali e separate sui successori e i predecessori durante una DFS.
  - Non considerano l'intera componente connessa.
- Metodo 3 (Albero di visita in DFS):
  - Questo metodo costruisce un albero DFS completo e conta tutti i nodi raggiungibili da `v0`.
  - Fornisce un conteggio più completo rispetto ai metodi 1 e 2, ma è comunque limitato alla struttura dell'albero DFS.
- Metodo 4 (Componente connessa):
  - Questo metodo è il più completo, in quanto considera tutti i nodi che sono nella stessa componente connessa di `v0`.
  - È il metodo più accurato per determinare tutti i nodi che possono essere raggiunti da `v0` indipendentemente dal percorso seguito.

In conclusione, il metodo 4 è generalmente il più utile per ottenere una visione completa della connettività del nodo `v0` nel grafo, mentre i metodi 1, 2 e 3 possono essere utili in contesti specifici in cui si vuole esaminare successori, predecessori o la struttura dell'albero DFS.

Lab 11

```
def searchPath(self, product_number):
      nodoSource = self.idMap[product_number]

      parziale = []

      self.ricorsione(parziale, nodoSource, 0)


      print("final", len(self._solBest), [i[2]["weight"] for i in
self._solBest])

def ricorsione(self, parziale, nodoLast, livello):
      archiViciniAmmissibili = self.getArchiViciniAmm(nodoLast,parziale)
      # caso uscita
      if len(archiViciniAmmissibili) == 0:
            if len(parziale) > len(self._solBest):
                  self._solBest = list(parziale)
                  print(len(self._solBest), [ii[2]["weight"] for ii in
self._solBest])
      # ritorsione
      for a in archiViciniAmmissibili:
            parziale.append(a)
            elf.ricorsione(parziale, a[1], livello + 1)
            parziale.pop()

def getArchiViciniAmm(self, nodoLast, parziale):
```

Questa funzione filtra gli archi vicini ammissibili del nodo `nodoLast` *in base a due criteri:* `isAscendent` *e* `isNovel`*.*

```
      archiVicini = self._grafo.edges(nodoLast, data=True)
      result = []
      for a1 in archiVicini:
            if self.isAscendent(a1,parziale) and self.isNovel(a1,parziale):
                  result.append(a1)
      return result

def isAscendent(self, e, parziale):
```

Questa funzione verifica se un arco è ascendente, cioè se il suo peso è maggiore o uguale all'ultimo arco nel percorso corrente (`parziale`*).*

```
      if len(parziale)==0:
            print("parziale is empty in isAscendent")
            return True
      return e[2]["weight"] >= parziale[-1][2][“weight"]

def isNovel(self, e, parziale):
```

Questa funzione verifica se un arco è nuovo, cioè non è già presente in `parziale` *né nel verso normale né in quello inverso.*

```
      if len(parziale)==0:
            print("parziale is empty in isnovel")
            return True
      e_inv = (e[1], e[0], e[2])
      return (e_inv not in parziale) and (e not in parziale)
```

FlightDelays

Questo codice è progettato per esplorare e valutare tutti i cammini possibili tra due nodi in un grafo, rispettando una lunghezza massima e cercando di massimizzare un valore obiettivo definito dal peso degli archi.

```
def getCamminoOttimo(self, v0, v1, t):
      self._bestPath = []
      self._bestObjFun = 0

      parziale = [v0]

      self._ricorsione(parziale, v1, t)

      return self._bestPath, self._bestObjFun

def _ricorsione(self, parziale, target, t):
      # Verificare che parziale sia una possibile soluzione
      # Verificare se parziale è meglio di best
      # esco

      if self.getObjFun(parziale)>self._bestObjFun and parziale[-1]==target:
            self._bestObjFun = self.getObjFun(parziale)
            self._bestPath = copy.deepcopy(parziale)

      if len(parziale) == t+1:
            return

      # Posso ancora aggiungere nodi.
      # prendo i vicini e provo ad aggiungere
      # ricorsione

      for n in self._grafo.neighbors(parziale[-1]):
            if n not in parziale:
                  parziale.append(n)
                  self._ricorsione(parziale, target, t)
                  parziale.pop()


def getObjFun(self, listOfNodes):
      objVal = 0
      for i in range(0, len(listOfNodes)-1):
            objVal += self._grafo[listOfNodes[i]][listOfNodes[i+1]]["weight"]

      return objVal
```

Baseball

```
def getPercorso(self, v0):
      self._bestPath = []
      self._bestObjVal = 0
      parziale = [v0]
      listaVicini = []
      for v in self._grafo.neighbors(v0):
            edgeV = self._grafo[v0][v]["weight"]
            listaVicini.append((v, edgeV))
      listaVicini.sort(key=lambda x: x[1], reverse=True)
```

#listaVicini viene ordinata in ordine decrescente in base al peso degli archi utilizzando sort e una lambda function come chiave.

```
      parziale.append(listaVicini[0][0])
      self._ricorsioneV2(parziale)
      parziale.pop()

      return self.getWeightsOfPath(self._bestPath)

def _ricorsione(self, parziale):
      # verifico se sol attuale è migliore del best
      if self._getScore(parziale) > self._bestObjVal:
            self._bestPath = copy.deepcopy(parziale)
            self._bestObjVal = self._getScore(parziale)

      # verifico se posso aggiungere un altro elemeneto
      for v in self._grafo.neighbors(parziale[-1]):
            edgeW = self._grafo[parziale[-1]][v]["weight"]
            if (v not in parziale and
                  self._grafo[parziale[-2]][parziale[-1]]["weight"] > edgeW):
                  parziale.append(v) #aggiungo e faccio ricorsione
                  self._ricorsione(parziale)
                  parziale.pop()

def _ricorsioneV2(self, parziale):
      # verifico se sol attuale è migliore del best
      if self._getScore(parziale) > self._bestObjVal:
            self._bestPath = copy.deepcopy(parziale)
            self._bestObjVal = self._getScore(parziale)

      # verifico se posso aggiungere un altro elemeneto
      listaVicini = []
      for v in self._grafo.neighbors(parziale[-1]):
            edgeV = self._grafo[parziale[-1]][v]["weight"]
            listaVicini.append( (v, edgeV) )

      listaVicini.sort(key=lambda x: x[1], reverse=True)

      for v1 in listaVicini:
            if (v1[0] not in parziale and
                  self._grafo[parziale[-2]][parziale[-1]]["weight"] > v1[1]):
                  parziale.append(v1[0])
                  self._ricorsioneV2(parziale)
                  parziale.pop() #aggiungo e faccio ricorsione

                  return
```

```
def _getScore(self, listOfNodes):

      if len(listOfNodes) == 1:
      return 0

      score = 0
      for i in range(0, len(listOfNodes)-1):
            score += self._grafo[listOfNodes[i]][listOfNodes[i+1]]["weight"]
      return score
```

Entrambe le funzioni di ricorsione verificano se il percorso attuale (parziale) ha un punteggio (_getScore) maggiore del migliore trovato finora (_bestObjVal). Se sì, aggiornano _bestPath e _bestObjVal.
_ricorsioneV2 potrebbe essere più efficiente in alcuni casi perché esplora prima i vicini con i pesi degli archi più alti, che potrebbero essere più promettenti.

- **_ricorsione**: Può essere più adatta quando si desidera esplorare tutti i percorsi possibili senza priorità sugli archi.
- **_ricorsioneV2**: Può essere più efficace quando si desidera esplorare prima i percorsi con gli archi più pesanti, nella speranza di trovare rapidamente un percorso ottimale, e si vuole limitare l'esplorazione aggiuntiva una volta trovato un vicino valido.

ITunes

```
def getSetAlbum(self, a1, dTOT):
      self._bestSet = None
      self._bestScore = 0
      connessa = nx.node_connected_component(self._graph, a1)
      parziale = set([a1])
      connessa.remove(a1)

      self._ricorsione(parziale, connessa, dTOT)

      return self._bestSet, self.durataTot(self._bestSet)

def _ricorsione(self, parziale, connessa, dTOT):
      #verificare se parziale è una sol ammissibile
      if self.durataTot(parziale) > dTOT:
            return

      #verificare se parziale è migliore del best
      if len(parziale) > self._bestScore:
            self._bestSet = copy.deepcopy(parziale)
            self._bestScore = len(parziale)

      #ciclo su nodi aggiungibili -- ricorsione
      for c in connessa:
            if c not in parziale:
                  parziale.add(c)
                  self._ricorsione(parziale, connessa, dTOT)
                  parziale.remove(c)

def durataTot(self, listOfNodes):
      dtot = 0
      for n in listOfNodes:
            dtot += n.totD
      return toMinutes(dtot)
```

Nyc-hotspot

```
def getCammino(self, target, substring):
      sources = self.getNodesMostVicini()
      source = sources[random.randint(0, len(sources)-1)][0]

      if not nx.has_path(self._graph, source, target):
            print(f"{source} e {target} non sono connessi.")
            return [], source

      self._bestPath = []
      self._bestLen = 0
      parziale = [source]

      self._ricorsione(parziale, target, substring)

      return self._bestPath, source

def _ricorsione(self, parziale, target, substring):
      if parziale[-1] == target:
      #devo uscire. ma prima controllo che sia una soluzione ottima
      if len(parziale) > self._bestLen:
            self._bestLen = len(parziale)
            self._bestPath = copy.deepcopy(parziale)
            return

      for v in self._graph.neighbors(parziale[-1]):
            if v not in parziale and substring not in v.Location:
                  parziale.append(v)
                  self._ricorsione(parziale, target, substring)
                  parziale.pop()
```

Lab 12

Dato il grafo costruito al punto precedente, si vuole identificare un percorso semplice e chiuso a peso massimo composto da esattamente N archi. Il valore di N deve essere inserito dall'utente tramite il campo apposito nell'interfaccia grafica. N deve essere almeno 2. A tal fine si identifichi la sequenza di vertici con le seguenti caratteristiche:

- Il primo e l'ultimo vertice della sequenza devono coincidere.
- I vertici intermedi non devono essere ripetuti
- La somma dei pesi degli archi percorsi deve essere massima.

Si visualizzi:

- La somma totale dei pesi degli archi percorsi nel percorso di peso massimo massimo
- Il percorso trovato come sequenza di archi, ciascuno dei quali nella forma:
  Nome_Retailer_1 —> Nome_ Retailer_2 : peso_arco

```
def computePath(self, N):
      self.path = []
      self.path_edge = []
      self.solBest = 0

      for r in self._ret_connected:
            partial = []
            partial.append(r)
            self.ricorsione(partial, N, [])

def ricorsione(self, partial, N, partial_edge):
      r_last = partial[-1]
      r_first = partial[0]

      #terminazione
      if len(partial_edge) == (N-1):
            if self._grafo.has_edge(r_last, r_first):
                  partial_edge.append((r_last, r_first,
self._grafo.get_edge_data(r_last, r_first)['weight']))
                  partial.append(r_first)
                  weight_path = self.computeWeightPath(partial_edge)
                  if weight_path > self.solBest:
                        self.solBest = weight_path + 0.0
                        self.path = partial[:]
                        self.path_edge = partial_edge[:]
                  partial.pop()
                  partial_edge.pop()
            return

      neighbors = list(self._grafo.neighbors(r_last))
      neighbors = [i for i in neighbors if i not in partial]
      for n in neighbors:
            partial_edge.append((r_last, n, self._grafo.get_edge_data(r_last, n)
['weight']))
            partial.append(n)

            self.ricorsione(partial, N, partial_edge)
            partial.pop()
            partial_edge.pop()

def computeWeightPath(self, mylist):
        weight = 0
        for e in mylist: weight += e[2] -> return weight
```

Terminazione della ricorsione:

- La ricorsione termina quando `partial_edge` contiene N–1 archi, il che significa che abbiamo visitato N nodi (incluso il nodo di partenza).
- Se esiste un arco che collega l'ultimo nodo (`r_last`) al nodo di partenza (`r_first`), viene aggiunto per formare un ciclo chiuso.
- Si calcola il peso totale del percorso con `self.computeWeightPath(partial_edge)`.
- Se il peso totale è maggiore del miglior percorso trovato finora (`self.solBest`), si aggiorna il miglior percorso e il suo peso.

Esplorazione dei vicini:

- Si ottengono i vicini del nodo corrente (`r_last`) che non sono già stati visitati (non sono in `partial`).
- Per ogni vicino valido, si aggiunge l'arco al percorso corrente (`partial_edge`) e si aggiunge il vicino al percorso (`partial`).
- Si chiama ricorsivamente la funzione `ricorsione` con il nuovo stato.
- Dopo la chiamata ricorsiva, si rimuove l'ultimo nodo e l'ultimo arco aggiunti (backtracking).

Ufo

```
def computePath(self):
      self.path = []
      self.path_edge = []

      for n in self.get_nodes():
            partial = []
            partial.append(n)
            self.ricorsione(partial, [])

def ricorsione(self, partial, partial_edge):
      n_last = partial[-1]

      neighbors = self.getAdmissibleNeighbs(n_last, partial_edge)

      # stop
      if len(neighbors) == 0:
            weight_path = self.computeWeightPath(partial_edge)
            if weight_path > self.solBest:
                  self.solBest = weight_path + 0.0
                  self.path = partial[:]
                  self.path_edge = partial_edge[:]
            return

      for n in neighbors:
            partial_edge.append((n_last, n, self._grafo.get_edge_data(n_last, n)
['weight']))
            partial.append(n)

            self.ricorsione(partial, partial_edge)
            partial.pop()
            partial_edge.pop()
```

```
def getAdmissibleNeighbs(self, n_last, partial_edges):
      all_neigh = self._grafo.edges(n_last, data=True)
      result = []
      for e in all_neigh:
            if len(partial_edges) != 0:
                  if e[2]["weight"] > partial_edges[-1][2]:
                  result.append(e[1])
            else:
                  result.append(e[1])
      return result

def computeWeightPath(self, mylist):
      weight = 0
      for e in mylist:
            weight += distance.geodesic((e[0].lat, e[0].lng), (e[1].lat,
e[1].lng)).km
      return weight
```

Gene-small

```
def searchPath(self, t):
      for n in self.get_nodes():
            partial = []
            partial_edges = []

            partial.append(n)
            self.ricorsione(partial, partial_edges, t)

      print("final", len(self.solBest), [i[2]["weight"] for i in self.solBest])

def ricorsione(self, partial, partial_edges, t):
      n_last = partial[-1]
      neigh = self.getAdmissibleNeighbs(n_last, partial_edges, t)

      # stop
      if len(neigh) == 0:
            weight_path = self.computeWeightPath(partial_edges)
            weight_path_best = self.computeWeightPath(self.solBest)
            if weight_path > weight_path_best:
                  self.solBest = partial_edges[:]
            return

      for n in neigh:
            partial.append(n)
            partial_edges.append((n_last, n, self.graph.get_edge_data(n_last,
n)))
            self.ricorsione(partial, partial_edges, t)
            partial.pop()
            partial_edges.pop()
```

```
def getAdmissibleNeighbs(self, n_last, partial_edges, t):
      all_neigh = self.graph.edges(n_last, data=True)
      result = []
      for e in all_neigh:
            if e[2]["weight"] > t:
                  e_inv = (e[1], e[0], e[2])
                  if (e_inv not in partial_edges) and (e not in partial_edges):
                      result.append(e[1])
      return result

def computeWeightPath(self, mylist):
      weight = 0
      for e in mylist:
            weight += e[2]['weight']

      return weight
```

TdP - siti utili

https://flet.dev/docs/controls -> sito documentazione flex

https://networkx.org/documentation/stable/reference/index.html -> sito documentazione nx

https://networkx.org/documentation/stable/reference/algorithms/traversal.html -> Graph traversal methods

https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html -> Cammini minimi

https://elite.polito.it/teaching/03fyz-tdp/lezioni -> sito corso