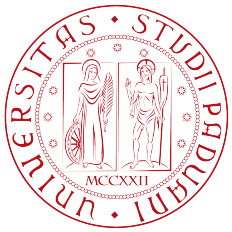
Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"
Corso di Laurea Magistrale in Informatica



Laboratorio di Algoritmi Avanzati

Traveling Salesman Problem

Studente:
Alessio Lazzaron
Matricola 1242456

Studente:
Matteo Marchiori
Matricola 1236329

Anno Accademico 2019–2020

INDICE

1 INTRODUZIONE

La relazione descrive gli algoritmi implementati da Alessio Lazzaron e Matteo Marchiori per il secondo laboratorio del corso di algoritmi avanzati.

1.1 DESCRIZIONE DEGLI ALGORITMI

Gli algoritmi implementati sono:

- Held Karp;
- Algoritmo 2-approssimato, con MST costruito usando l'algoritmo di Prim;
- Euristica costruttiva Cheapest Insertion.

In seguito riportiamo lo pseudocodice degli algoritmi, e la parte saliente del codice implementato in Python in modo da confrontarli facilmente.

1.1.1 Algoritmo di Held Karp

```
Held-Karp(G=(V,E), S, v):
    if(S = {v}) return w[v,0]
    if(d[v,S] is not NULL) return d[v,S]
    mindist = inf
    minprec = NULL
    for u in S\{v} do:
        dist = Held-Karp(u, S\{v})
        if dist + w[u,v] < mindist:
            mindist = dist + w[u,v]
            minprec = u
    d[s,V] = mindist
    p[s,V] = minprec
    return mindist
```

```
def held_karp(start, v, S: list, d_dict, p_dict):
    global stop
    if len(S) == 1 and v in S:
        return v.find_edge(start).weight()
    elif (str(v.name()+1), str(S)) in d_dict:
        return d_dict[(str(v.name()+1), str(S))]
    else:
        mindist = float("Inf")
        minprec = None
        S2 = list(S)
        S2.remove(v)
        for i in range(len(S2)):
            dist = held_karp(start, S2[i], S2, d_dict, p_dict)
            uv_weight = S2[i].find_edge(v).weight()
            if(dist + uv_weight < mindist):
                mindist = dist + uv_weight
                minprec = S2[i]
            if stop:
                break
        d_dict[(str(v.name()+1), str(S))] = mindist
        p_dict[(str(v.name()+1), str(S))] = minprec
        return mindist
```

Per tenere memoria delle distanze minime e dei nodi predecessori abbiamo utilizzato dizionari di python, in modo da mantenere l'indicizzazione attraverso (nodo destinazione , lista dei nodi coperti) vista a lezione. Dato che l'algoritmo ha una complessità $O(2^n)$ è stata inserito un meccanismo per fermare il calcolo dopo un numero di secondi specificabile.

1.1.2 Algoritmo 2-approssimato

```
Preorder(v):
    print(v)
    if(internal(v)):
        foreach u in children(v) do:
            preorder(u)

Approx_t_tsp(G=(V,E), c):
    V = {v1...vn}
    root = v1
    T = Prim(G,c,root)
    <v1...vn> = Preorder(root)
    return <v1...vn, v1>
```

```
def preorder(preordered, v):
    preordered.append(v)
    for child in v.children():
        preorder(preordered, child)


def approx_t_tsp(graph, s):
    prim(graph, s)
    nodes = graph.get_graph()
    preordered = []
    preorder(preordered, nodes[s])
    preordered.append(nodes[s])
    return preordered
```

In questo algoritmo costruiamo una lista dei nodi di un MST trovato usando Prim in ordine prefisso, aggiungendo al termine della lista il nodo radice. Funziona perché vale la disuguaglianza triangolare, ovvero per arrivare da un nodo sorgente a un nodo destinazione la soluzione migliore include l'arco che li collega direttamente. Osservando la soluzione peggiore di TSP trovata con l'algoritmo, si vede che non può essere peggiore rispetto a prendere due volte il costo della soluzione ottima. Quindi l'algoritmo di 2-approssimazione è corretto.

1.1.3 Algoritmo Cheapest Insertion

```
Cheapest_insertion(G=(V,E)):
    sol = (0,j,0)
    do:
        (k,i)+(k,j)-(i,j) minimize cost
        sol = sol + (k,i) + (k,j) - (i,j)
    while(exists k outside sol)
```

```
def cheapest_insertion(G:Graph):
edges = G.edges()
nodes = list(G.nodes())
edges_sol = {}
weight_sol = 0
zero_n = nodes.pop(0)

while len(nodes) != 0:
    local_min = float("inf")
    local_edge = [None]*3
    local_node = None
    if len(edges_sol) == 0:
        for i in range(len(nodes)):
            val = edges[zero_n.name(), nodes[i].name()]
            if val.weight() < local_min:
                local_min = val.weight()
                local_node = nodes[i]
                local_edge[0] = val
    else:
        for k in range(len(nodes)):
            for idx, val in enumerate(edges_sol.values()):
                ik = edges[nodes[k].name(), val.nodes()[0].name()]
                jk = edges[nodes[k].name(), val.nodes()[1].name()]
                ij = val

                to_minimized = ik.weight() + jk.weight() - ij.weight()
                if to_minimized < local_min:
                    local_min = to_minimized
                    local_node = nodes[k]
                    local_edge = ik, jk, val

    nodes.remove(local_node)
    if not local_edge[1]:
        weight_sol += local_min*2
        nodes_e = local_edge[0].nodes()
        edges_sol[nodes_e[0].name(), nodes_e[1].name()] = local_edge[0]
        edges_sol[nodes_e[1].name(), nodes_e[0].name()] =\
            Edge(nodes_e[1], nodes_e[0], local_edge[0].weight())
```

```
        else:
            weight_sol += local_edge[0].weight() + local_edge[1].weight()\
                - local_edge[2].weight()
            nodes_0 = local_edge[0].nodes()
            nodes_1 = local_edge[1].nodes()
            nodes_2 = local_edge[2].nodes()
            edges_sol[nodes_0[0].name(), nodes_0[1].name()] = local_edge[0]
            edges_sol[nodes_1[0].name(), nodes_1[1].name()] = local_edge[1]
            del edges_sol[nodes_2[0].name(), nodes_2[1].name()]

    return weight_sol
```

Questo algoritmo di euristica costruttiva parte da un nodo radice e sceglie l'arco incidente su questo nodo che sia di costo minimo. Dopodiché per ogni nodo k, non presente nella soluzione del cammino minimo, vengono scelti due archi: ik e jk, tali che i nodi i, j appartengono alla solucione e tali archi minimizzano la somma tra i lori pesi e la differenza tra il peso dell'arco ij.

1.1.4 Scelte implementative comuni

Per tutti gli algoritmi implementati abbiamo cercato di usare, per quanto possibile, la definizione di oggetti e la definizione di metodi previsti da python, in modo da mantenere per quanto possibile il codice comprensibile.

2 RISULTATI

In questa sezione mostriamo i risultati ottenuti dai tre algoritmi, in termini di tempo impiegato, soluzione ed errore commesso rispetto all'ottimo.

2.1 TABELLA

Istanza	Held-Karp			Cheapest Insertion			2-approssimato		
	Sol	Tempo (s)	Errore	Sol	Tempo (s)	Errore	Sol	Tempo (s)	Errore
berlin52	18904	180.00	150.6%	8988	0.05	19.17%	12132	0.003	60.86%
burma14	3323	6.01	-	3588	0.001	7.974%	4690	0.0005	41.14%
ch150	49470	180.03	657.8%	7878	1.43	20.68%	11640	0.02	78.31%
d493	112243	180.28	220.7%	39760	57.54	13.59%	58166	0.28	66.18%
dsj1000	555572539	181.20	2800%	22290672	526.06	19.46%	31809536	1.01	70.47%
eil51	1093	180.00	156.6%	479	0.05	12.44%	718	0.003	68.54%
gr202	56641	180.05	41.03%	46480	3.91	15.73%	65246	0.04	62.46%
gr229	176922	180.06	31.44%	153896	5.42	14.34%	227954	0.06	69.35%
kroA100	175122	180.02	722.9%	24175	0.40	13.59%	37462	0.01	76.03%
kroD100	152902	180.02	618.1%	25155	0.42	18.13%	37104	0.01	74.25%
pcb442	215368	180.23	324.1%	60766	41.31	19.67%	92636	0.20	82.43%
ulysses16	6859	35.65	-	7368	0.001	7.420%	9080	0.0004	32.38%
ulysses22	8649	180.01	23.33%	7845	0.004	11.86%	9320	0.0008	32.90%

Tabella 1: Risultati

2.2 RISULTATI

Dai risultati ottenuti si può vedere che malgrado l'algoritmo di Held Karp sia un algoritmo esatto richiede molto tempo per trovare il risultato cercato. Troncando il tempo di esecuzione a tre minuti, la soluzione esatta viene trovata solamente per i grafi più piccoli, composti al massimo da 16 nodi. Gli algoritmi cheapest insertion e 2-approssimazione invece impiegano un tempo accettabile su tutti i grafi (entro i 10 minuti impiegati da cheapest insertion per il grafo più grande). Questo è dovuto alla complessità molto inferiore rispetto a quella di Held Karp, che è esponenziale. L'algoritmo migliore, rispetto all'errore di approssimazione, risulta essere stranamente cheapest insertion, pur non promettendo un bound superiore garantito. Da quanto visto a lezione tale limite dovrebbe essere uguale a quello dell'algoritmo di 2-approssimazione, che effettivamente risulta rispettato. L'algoritmo più veloce nel calcolo dell'approssimazione è quello di 2-approssimazione, che commette però un errore più elevato rispetto agli altri due algoritmi (in realtà l'errore più elevato è commesso da Held Karp con interruzione, ma senza considerare un'interruzione ai tre minuti tale algoritmo fornirebbe un risultato esatto).

3 CONCLUSIONI

3.1 RISULTATI OTTENUTI

Dai risultati ottenuti possiamo dire che l'analisi della complessità dei vari algoritmi rispecchia i tempi impiegati per il calcolo dei risultati. Held Karp impiega molto tempo, pur essendo un algoritmo esatto, mentre cheapest insertion e 2-approssimazione forniscono un'approssimazione del risultato. Considerati i tempi impiegati e le soluzioni approssimate, si può dire che avendo tempo infinito a disposizione l'algoritmo preferibile è Held Karp. Mettendo un bound a cheapest insertion, pur essendo un algoritmo euristico, tale algoritmo è preferibile rispetto a quello di 2-approssimazione, per un errore minore commesso nel calcolo del risultato. Nel caso sia necessario avere un risultato approssimato con garanzia del bound di 2-approsimazione nel minor tempo possibile, l'algoritmo migliore risulta essere quello di 2-approssimazione.