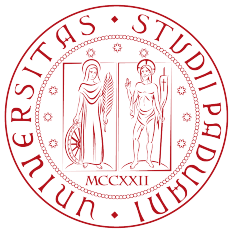
Università degli Studi di Padova

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



Laboratorio di Algoritmi Avanzati

Minimum Cut

Studente:
Alessio Lazzaron
Matricola 1242456

Studente:
Matteo Marchiori
Matricola 1236329

Anno Accademico 2019–2020

INDICE

1 INTRODUZIONE

La relazione descrive l'algoritmo implementato da Alessio Lazzaron e Matteo Marchiori per il terzo laboratorio del corso di algoritmi avanzati.

1.1 DESCRIZIONE DELL'ALGORITMO

L'unico algoritmo implementato per questo laboratorio è quello di Karger. In seguito poniamo lo pseudocodice e la parte interessante del codice in python per un rapido confronto, e alcune scelte fatte durante l'implementazione.

1.1.1 Algoritmo di Karger

```
Full_contraction(G=(V,E)):
    for i = 1 to |V| - 2 do:
        e = random(E)
        G' = (V, E') = G/e
        V = V'
        E = E'
    return |E|

Karger(G=(V,E), k):
    min = \(\infty\)
    for i = 1 to k do:
        t = Full_contraction(G)
        if t < min:
            min = t
    return min
```

```
def full_contraction(graph):
    nodes = graph.nodes()
    not_removed = list(nodes)
    for i in range(len(nodes)-2):
        src_index = random.randint(0, len(not_removed)-1)
        src = not_removed[src_index]
        adjacents_src = src.adjacents()
        dest = random.choice(adjacents_src)
        dest_index = dest.index()
        adjacents_dest = dest.adjacents()
        newname = graph.inc_lastnode()
        adjacents_src = [
            adjacent for adjacent in adjacents_src if adjacent != dest
        ]
        adjacents_dest = [
            adjacent for adjacent in adjacents_dest if adjacent != src
        ]
        newadjacents = adjacents_src + adjacents_dest
        newnode = Node(newname, dest_index)
        newnode.set_adjacents(newadjacents)
        for node in newadjacents:
            adjacents = node.adjacents()
            for i, adjacent in enumerate(adjacents):
                if adjacent is src or adjacent is dest:
                    adjacents[i] = newnode
        nodes[src_index] = None
        nodes[dest_index] = newnode
        del not_removed[src_index]

    mincut = 0
    for node in not_removed:
        if node is not None:
            for adjacent in node.adjacents():
                if adjacent is not None:
                    mincut += 1
            break
    return mincut
```

```
def karger(graph, k, ottimo):
    mincut = float("inf")
    original = copy.deepcopy(graph)
    time_fc = 0
    time_discovery = 0
    found = False
    for _ in range(k):
        fc_start = time.time()
        fc = full_contraction(graph)
        if fc == ottimo and not found:
            time_discovery = time.time()
            found = True
        elif fc < ottimo:
            time_discovery = time.time()
            ottimo = fc
        time_fc += time.time() - fc_start
        if fc < mincut:
            mincut = fc
        graph = copy.deepcopy(original)
    time_fc = time_fc / k
    return mincut, time_fc, time_discovery
```

Nell'algoritmo full_contraction la contrazione di due nodi avviene creando un nuovo nodo nel grafo che unisce due nodi presi a caso, il primo dai nodi non ancora eliminati, il secondo tra i possibili nodi adiacenti al primo scelto (di modo da avere almeno un arco esistente tra questi). Il nuovo nodo ha come nodi adiacenti i nodi adiacenti presi dai nodi scelti, e viene memorizzato al posto del secondo nodo scelto, sovrascrivendolo, mentre il primo viene impostato a None, eliminandolo. Alla fine la procedura restituisce il taglio trovato.

Il secondo algoritmo viene fatto ciclare k volte, dove k è determinato con la formula vista a lezione. Vengono determinati i tempi richiesti e prima di chiamare full_contraction il grafo viene impostato alla versione fornita in input (pertanto vi sarà uno scarto temporale che dipende dalla dimensione del grafo e da k nella stima dei tempi dell'algoritmo di karger).

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 dall'algoritmo, ovvero i tempi medi impiegati per full contraction e karger, il discovery time, la soluzione trovata, la soluzione attesa e l'errore relativo.

2.1 GRAFICI

2.1.1 Full contraction

Il presente grafico illustra i tempi di calcolo della procedura Full_Contraction sui grafi del dataset al variare del numero di vertici. Confrontiamo con la complessità asintotica stimata di $O(n^2)$.

2.1.2 Karger

Il presente grafico illustra i tempi di calcolo dell'algoritmo di Karger sui grafi del dataset al variare del numero di vertici. La probabilità di errore è uguale a $\frac{1}{n}$ di sbagliare per grafi inferiori a 75 nodi. Tale scelta è dovuta a motivi di tempo, come illustrato in sezione 3 a pagina 6. Confrontiamo con la complessità asintotica stimata di $O(n^4 log(n))$.

2.1.3 Discovery time

Il presente grafico illustra i tempi di calcolo dell'algoritmo di Karger per trovare la soluzione ottima per la prima volta. Confrontiamo i risultati con il tempo di esecuzione complessivo.

2.2 TABELLA

Istanza	Full contraction	Karger	Discovery time	Risultato	Ottimo	Errore relativo

Tabella 1: Risultati

2.3 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 ORIGINALITÀ

In questa sezione vengono presentate le originalità che abbiamo inserito nella nostra implementazione dell'algoritmo.

È possibile ottenere probabilità di insuccesso diverse nell'algoritmo cambiando il parametro d passato da linea di comando. Di default il parametro è impostato a 1 per avere una probabilità di insuccesso pari a $\frac{1}{n}$, dove n è il numero dei nodi del grafo.

Nell'implementazione sviluppata abbiamo impostato k a un valore inferiore quando la taglia del grafo supera i 75 nodi, ovvero a una costante in cui n vale 75, ottenendo come k il valore $\frac{75^2}{2} \cdot ln(75)$, questo per evitare tempistiche improponibili per l'esecuzione dell'algoritmo.

4 CONCLUSIONI

4.1 RISULTATI OTTENUTI

Dai risultati ottenuti è visibile come i tempi rispecchino la complessità dell'algoritmo stimata.

Per avere una stima più corretta dei tempi medi impiegati dall'algoritmo di Karger sarebbe necessario togliere il tempo impiegato per la copia del grafo, che potrebbe influire negativamente nelle istanze di dimensione maggiore.