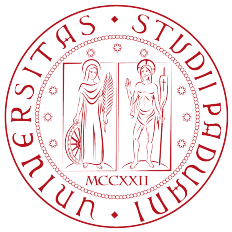
Università degli Studi di Padova

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



Laboratorio di Algoritmi Avanzati

Implementazione di tre algoritmi per MST

Studente:
Alessio Lazzaron
Matricola

Studente:
Matteo Marchiori
Matricola 1236329

Anno Accademico 2019–2020

INDICE

ELENCO DELLE FIGURE

1 INTRODUZIONE

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

1.1 DESCRIZIONE DEGLI ALGORITMI

Gli algoritmi implementati sono:

- Prim, implementato con Heap;
- Kruskal nella sua implementazione "naive" di complessità O(mn);
- Kruskal implementato con Union-Find

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 Prim

```
Prim(G,s)
    foreach u in V:
        key[u] = infinity
        parent[u] = null
    key[s] = 0
    Q = V
    while Q not empty:
        u = extractMin(Q)
        foreach v adjacent_to u:
            if v in Q and w(u,v) < key[v]:
                parent[v] = u
                key[v] = w(u,v)
    return V\Q
```

```
def prim(graph, s):
    mst_weight = 0
    adjacency_list = graph.get_graph()
    adjacency_list[s].set_key(0)
    heap_keys = Heap()
    for node in adjacency_list:
        heap_keys.push(node)
    while len(heap_keys) !=0:
        u = heap_keys.pop()
        adjacents = adjacency_list[u.name()].edges()
        for edge in adjacents:
            v = edge.node()
            if( v.present() and edge.weight() < v.key()):
                v.set_parent(u)
                v.set_key(edge.weight())
                heap_keys._orderup(v._position)
        mst_weight += u.key()
    return mst_weight
```

Abbiamo implementato la struttura Heap rappresentandola come una lista di nodi del grafo, che possiedono una chiave per cui vale la proprietà di min heap. I nodi possiedono inoltre un campo che identifica la posizione all'interno dello heap, per poterlo riordinare quando vengono spostati, e un campo present per verificare se il nodo in questione fa parte dello heap o meno. Per rappresentare il grafo abbiamo usato una lista di adiacenza, in cui ogni nodo ha una lista di archi incidenti in esso, e ogni arco mantiene l'informazione relativa al nodo opposto.

1.1.2 Algoritmo di Kruskal "naive"

```
Kruskal(G)
    A = {}
    sort edges of G by cost (mergesort)
    for each edge e in nondecreasing order of cost do:
        if A U {e} is acyclic then:
            A = A U {e}
    return A
```

```
def kruskal(graph, s):
graph.ordinaLati()
mst_weight = 0
visitati = [False] * graph.num_vertici()
componente = 0
for edge in graph.get_edges():
    node1 = edge.nodes()[0]
    node2 = edge.nodes()[1]
    edge.set_mst(True)
    if node1 == node2:
        edge.set_mst(False)
    elif not visitati[node1.name()] and not visitati[node2.name()]:
        componente+=1
        node1.set_component(componente)
        node2.set_component(componente)
    elif not visitati[node1.name()]:
        node1.set_component(node2.component())
    elif not visitati[node2.name()]:
        node2.set_component(node1.component())
    else:
        if node1.component() != node2.component():
            fix_component(graph, node1)
        else:
            edge.set_mst(False)
    visitati[node1.name()] = True
    visitati[node2.name()] = True
    if edge.mst():
         mst_weight += edge.weight()
return mst_weight
```

Per ordinare i lati in modo efficiente in ordine crescente di costo abbiamo implementato un mergesort nell'oggetto Graph. In questa versione dell'algoritmo verifichiamo su quali componenti connesse incide un lato del minimum spanning tree in costruzione. Nel caso il lato non incida in nessuna componente, viene aggiunta una nuova componente connessa. Altrimenti si verifica che i due nodi su cui il lato incide non appartengano alla medesima componente connessa. In tal caso il lato formerebbe un ciclo, e non viene aggiunto. Negli altri casi il lato viene aggiunto, e in caso vengano connesse due componenti connesse distinte viene usato DFS per ridurle a un'unica componente connessa, aggiornando i nodi coinvolti.

1.1.3 Algoritmo di Kruskal

```
Kruskal(G)
    A = {}
    U = initialize(V)
    sort edges of E by cost
    for each edge = (v,w) in non decreasing order of cost do:
        if find(U,v) != find(U,w):
            A = A U {(v,w)}
            Union(U,v,w)
    return A
```

```
def kruskal(graph, s):
mst_weight = 0
mst = Graph(graph.num_vertici())
union_find = UnionFind(graph.num_vertici())
graph.ordinaLati()
for edge in graph.get_edges():
    x = union_find.find(edge.nodes()[0].name())
    y = union_find.find(edge.nodes()[1].name())
    if x[0] != y[0]:
        s1 = mst.add_node(edge.nodes()[0].name())
        d1 = mst.add_node(edge.nodes()[1].name())
        mst.add_edge(s1, d1, edge._weight)
        union_find.union(s1.name(), d1.name())
        mst_weight += edge.weight()
return mst_weight
```

Per rappresentare la struttura union find abbiamo usato una lista di interi, rappresentati i nomi dei nodi. L'implementazione rispetta quanto visto a lezione per quanto concerne la complessità e le operazioni effettuate in tale struttura (init, union e find).

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 DIAGRAMMI E RISULTATI

In questa sezione commentiamo alcuni diagrammi che illustrano i risultati ottenuti dai tre algoritmi implementati in termini di tempo a fronte della taglia del grafo.

2.1 DIAGRAMMI

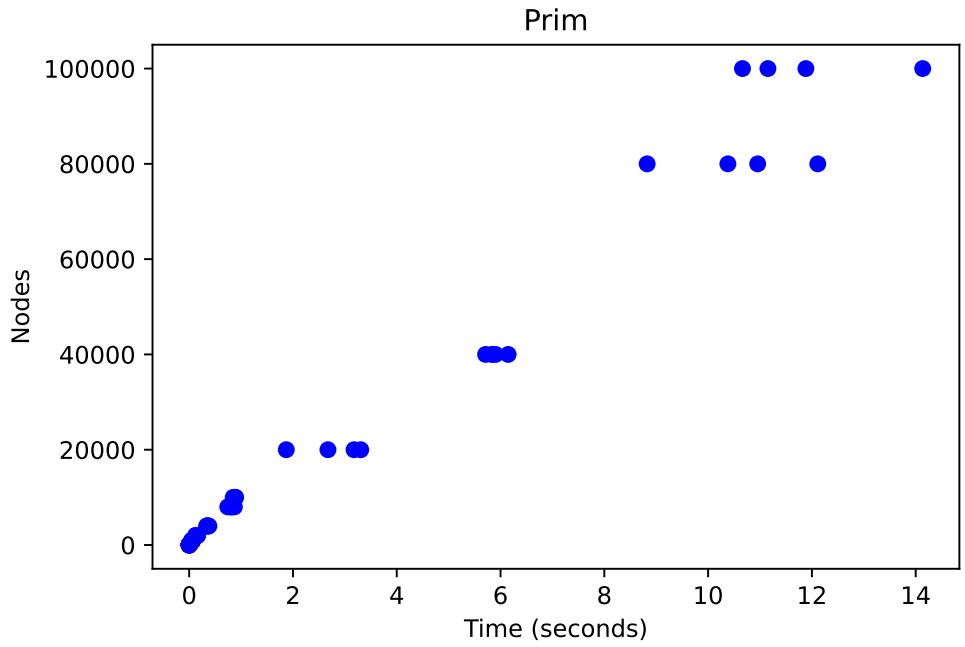


Figura 1: Diagramma con tempi impiegati per Prim

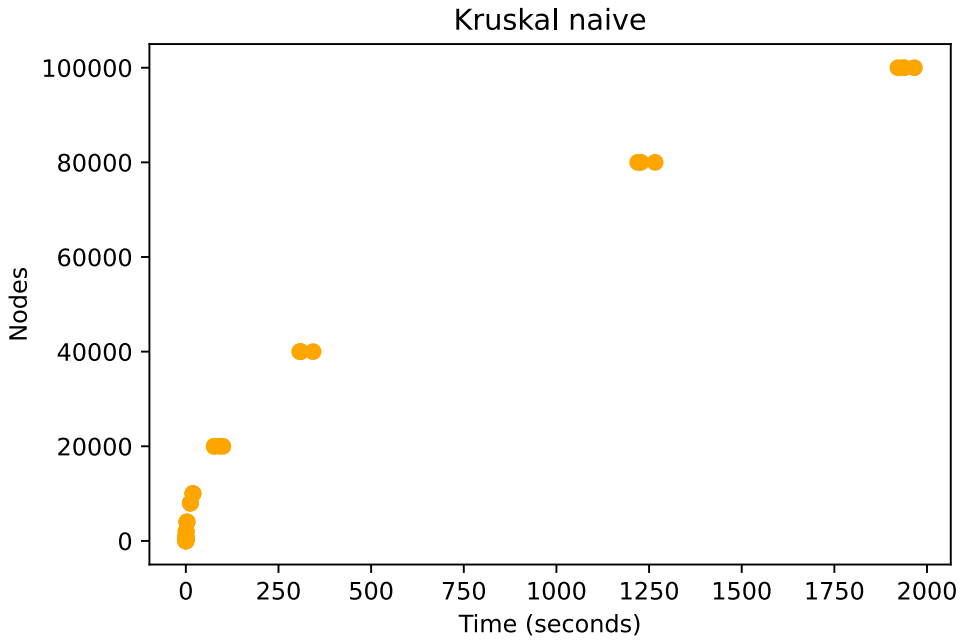


Figura 2: Diagramma con tempi impiegati per Kruskal "naive"

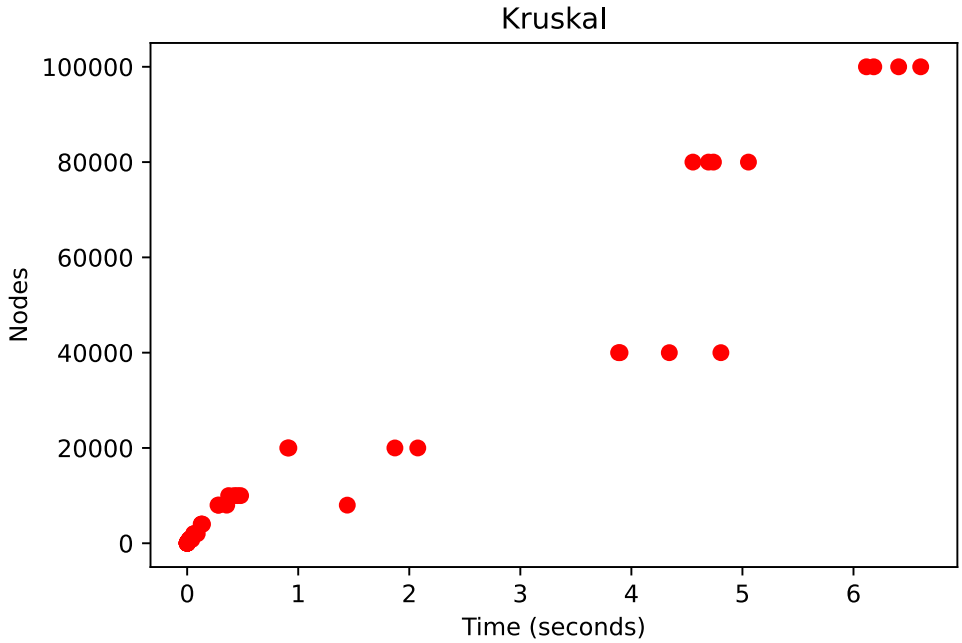


Figura 3: Diagramma con tempi impiegati per Kruskal con Union-Find

2.2 RISULTATI

È chiaramente visibile dai diagrammi che rappresentano le tempistiche impiegate che Kruskal naive si comporta peggio rispetto agli altri algoritmi per la ricerca di un MST. Nell'implementazione sviluppata la complessità è inferiore a O(mn), ma superiore a quella di Prim e Kruskal con Union Find. Trattiamo l'ottimizzazione introdotta nella sezione successiva.

Gli altri due algoritmi sono comparabili a livello di complessità, infatti i tempi impiegati sono dello stesso ordine di grandezza.

Pur essendo Prim e Kruskal con Union Find caratterizzati dalla stessa complessità computazionale (O(mlog(n))), per quanto misurato il secondo algoritmo si comporta meglio e impiega metà del tempo a parità di dimensione del grafo. Questo potrebbe essere dovuto alla struttura usata per rappresentare il grafo (lista di adiacenza nel caso di Prim, lista di adiacenza e singoli lati nel caso di Kruskal con Union Find).

3 ORIGINALITÀ

Qui vengono presentate alcune specialità usate nell'implementazione.

3.1 KRUSKAL NAIVE

Per quanto riguarda l'implementazione di Kruskal naive, per cercare la ciclicità ci siamo basati sulle componenti connesse nella costruzione di un mst partendo da un grafo, principio simile a quello adottato da Kruskal con Union Find. Quando viene aggiunto un lato al MST, si controllano i nodi su cui tale lato incide. Se tali nodi fanno parte della stessa componente connessa, il lato viene rimosso, altrimenti formerebbe un ciclo. In caso contrario viene aggiunto, e si aggiornano le componenti connesse di alcuni nodi, in modo da ottenerne una sola.

La ricerca di ciclicità è corretta per costruzione.

La complessità sembra inferiore a O(mn), perché vengono aggiornati solamente i nodi del MST con componente connessa errata e solamente quando essa è errata. Resta in ogni caso superiore a Prim e Kruskal con Union Find, perché non viene fatta alcuna considerazione sul modo in cui viene aggiornata tale componente. Nel caso peggiore, ovvero quando il grafo dato fosse un MST, e gli archi avessero una disposizione tale da essere aggiunti in modo da aggiornare sempre la componente connessa di n-1 nodi, la complessità risulterebbe O(mn/2). Senza considerare la costante l'algoritmo implementato è O(mn).

4 CONCLUSIONI

4.1 RISULTATI OTTENUTI

Dai risultati ottenuti possiamo dire che Kruskal con Union Find e Prim hanno una complessità molto inferiore rispetto a Kruskal naive nella ricerca di un MST.