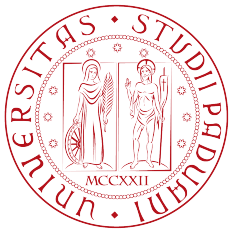
Università degli Studi di Padova

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

ELENCO DELLE TABELLE

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. Per rendere più facile la compressione della relazione, intenderemo con Kruskal-naive l'implementazione dell'algoritmo di Kruskal con complessità O(mn) mentre ci riferiremo semplicemente con Kruskal al medesimo all'algoritmo che utilizza la struttura dati Union-Find.

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
```

L'algoritmo prevede l'ordinamento dei lati in modo crescente, per farlo efficiente abbiamo implementato un metodo nella classe Graph che utilizza l'algoritmo Merge Sort per ordinarli. In questa versione dell'algoritmo verifichiamo su quali componenti connesse incide un lato del minimum spanning tree in costruzione. Nel caso in cui il lato non incida in nessuna componente connessa, quest'ultimo andrà a creare una nuova componente connessa. Invece, se il lato che si cerca di aggiungere va ad incidere su due nodi che appartengono alla medesima componente connessa allora si formerebbe un ciclo e quindi il lato non viene aggiunto. Mentre

se vengano connesse due componenti connesse distinte viene usato un approccio a DFS per ridurle ad un'unica componente connessa. In tutti gli altri casi il lato viene semplicemente aggiunto.

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, che rappresentano i nomi dei nodi. La nostra implementazione, di tale struttura, rispetta quanto visto a lezione sia per complessità, sia per le operazioni permesse (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 vengono mostrati i diagrammi che illustrano i risultati ottenuti dai tre algoritmi implementati in termini di tempo a fronte della taglia del grafo.

2.1 DIAGRAMMI

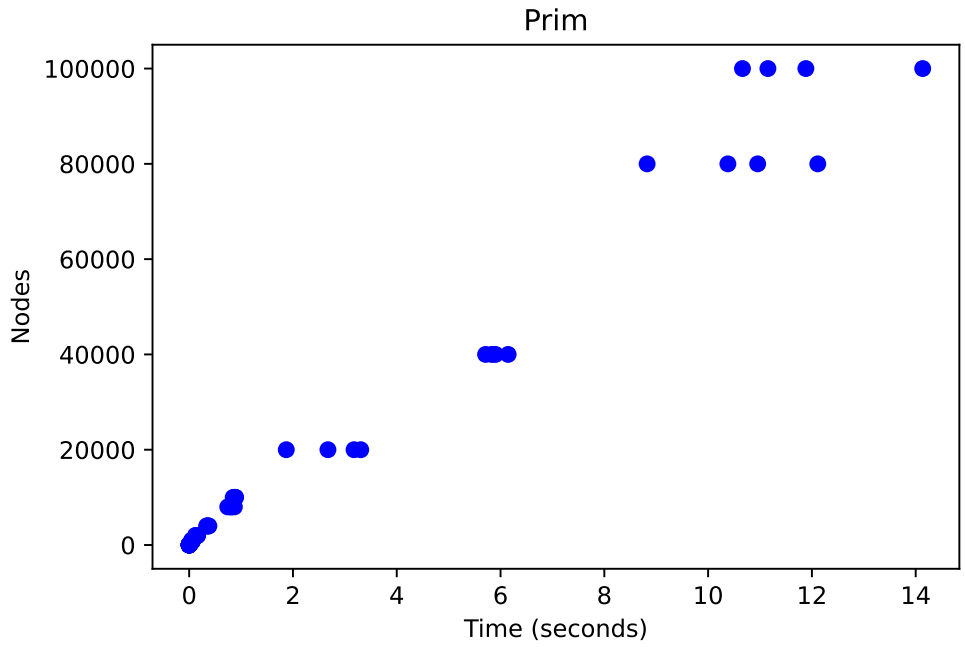


Figura 1: Diagramma con tempi impiegati per Prim

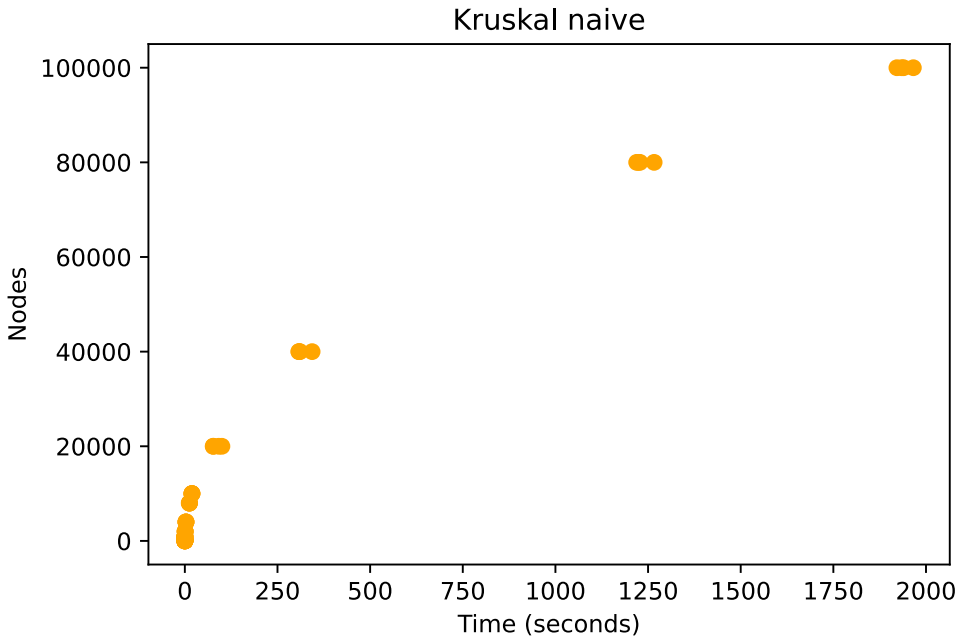


Figura 2: Diagramma con tempi impiegati per Kruskal-naive

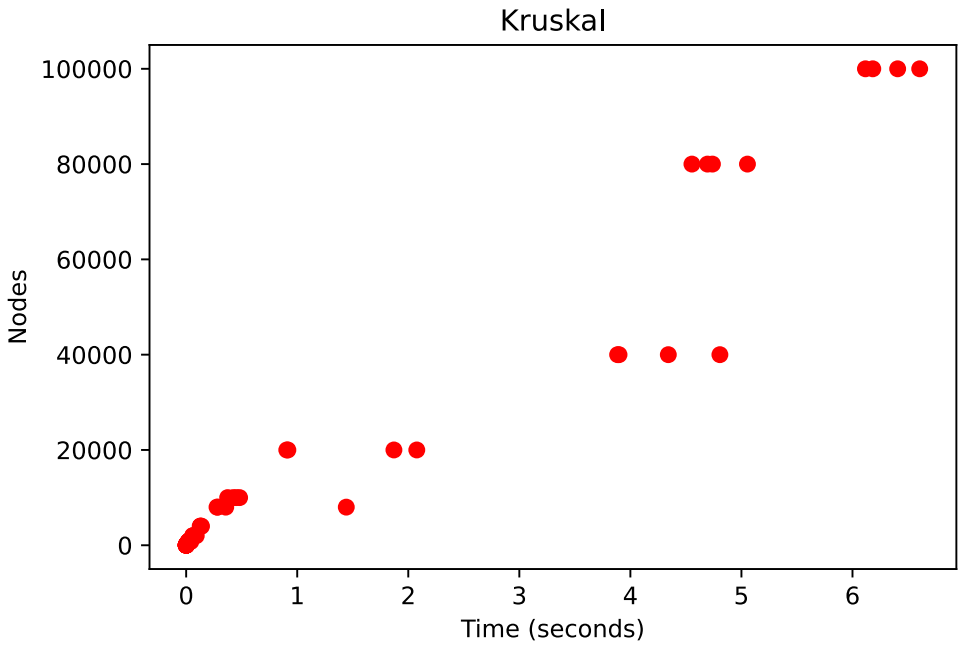


Figura 3: Diagramma con tempi impiegati per Kruskal

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 Kruska. Tratteremo l'ottimizzazione introdotta nella sezione successiva.

Pur essendo Prim e Kruskal 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).

3 ORIGINALITÀ

In questa sezione vengono presentate le originalità che abbiamo inserito nella nostra implementazione dei tre algoritmi.

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, allora il lato non viene aggiunto perchè altrimenti formerebbe un ciclo. In caso contrario viene aggiunto, e si aggiornano le componenti connesse di alcuni nodi, in modo da ottenerne una sola.

Per come l'abbiamo implementata, la ricerca di ciclicità è corretta per costruzione.

Inoltre sembra che la complessità sia 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, 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.

4.2 PESI DEGLI MST

Di seguito riportiamo, per ogni grafo compreso nel dataset, il peso complessivo del rispettivo minimum spanning tree e il tempo di esecuzioni per i tre algoritmi.

Grafo	Peso MST	Tempo Prim	Tempo Kruscal naive	Tempo Kruskal Union Find
01 10	29316	0.000 229	0.000 315	0.000 175
02 10	2126	0.000 268	0.000 368	0.000 211
03 10	-44765	0.000 304	0.000 375	0.000 254
04 10	20360	0.000 262	0.000 244	0.000 171
05 20	-32021	0.000 355	0.000 421	0.000 585
06 20	18596	0.000 551	0.000 831	0.000 592
07 20	-42560	0.000 711	0.000 859	0.000 435
08 20	-37205	0.000 607	0.001 024	0.000 642
09 40	-122078	0.000 820	0.000 950	0.000 830
10 40	-37021	0.001 563	0.001 073	0.000 838
11 40	-79570	0.003 746	0.000 943	0.000 703
12 40	-79741	0.000 866	0.000 965	0.000 835
13 80	-139926	0.002 844	0.002 393	0.003 041
14 80	-211345	0.004 792	0.002 391	0.001 631
15 80	-110571	0.001 800	0.002 930	0.001 593
16 80	-233320	0.002 836	0.001 965	0.001 661
17 100	-141960	0.002 923	0.004 229	0.002 216
18 100	-271743	0.002 424	0.003 495	0.002 021
19 100	-288906	0.003 076	0.003 920	0.002 208
20 100	-232178	0.003 444	0.003 225	0.001 975
21 200	-510185	0.005 745	0.010 034	0.005 886

Grafo	Peso MST	Tempo Prim	Tempo Kruscal naive	Tempo Kruskal Union Find
22 200	-515136	0.006 170	0.010 315	0.004 440
23 200	-444357	0.013 323	0.007 686	0.004 470
24 200	-393278	0.013 813	0.011 958	0.004 324
25 400	-1122919	0.030 481	0.038 656	0.010 893
26 400	-788168	0.027 348	0.031 573	0.009 061
27 400	-895704	0.022 651	0.032 715	0.011 281
28 400	-733645	0.022 417	0.036 082	0.009 439
29 800	-1541291	0.061 166	0.158 504	0.019 831
30 800	-1578294	0.060 090	0.131 379	0.020 724
31 800	-1675534	0.060 243	0.152 749	0.019 821
32 800	-1652119	0.044 129	0.122 067	0.042 726
33 1000	-2091110	0.060 992	0.176 050	0.040 210
34 1000	-1934208	0.052 957	0.222 203	0.031 411
35 1000	-2229428	0.050 793	0.230 049	0.031 874
36 1000	-2359192	0.056 321	0.174 618	0.039 273
37 2000	-4811598	0.126 127	0.784 476	0.075 792
38 2000	-4739387	0.124 496	0.715 397	0.090 823
39 2000	-4717250	0.162 983	0.719 663	0.058 472
40 2000	-4537267	0.158 729	0.731 537	0.060 427
41 4000	-8722212	0.366 342	2.905 543	0.130 016
42 4000	-9314968	0.379 759	2.883 613	0.126 211
43 4000	-9845767	0.337 094	2.959 421	0.130 238
44 4000	-8681447	0.346 852	3.309 770	0.138 210
45 8000	-17844628	0.870 938	12.650 625	0.283 904
46 8000	-18800966	0.807 914	12.449 594	0.356 206
47 8000	-18741474	0.809 766	11.735 250	0.277 730
48 8000	-18190442	0.743 533	12.396 922	1.441 794
49 10000	-22086729	0.848 342	19.195 808	0.423 291
50 10000	-22338561	0.877 863	19.124 557	0.452 050
51 10000	-22581384	0.895 806	19.175 858	0.481 175
52 10000	-22606313	0.885 038	19.089 805	0.373 543
53 20000	-45978687	1.871 198	76.550 140	0.915 692
54 20000	-45195405	2.673 921	76.676 509	0.905 859
55 20000	-47854708	3.176 991	100.014 610	1.870 719

Grafo	Peso MST	Tempo Prim	Tempo Kruscal naive	Tempo Kruskal Union Find
56 20000	-46420311	3.306 051	91.586 885	2.076 618
57 40000	-92003321	5.830 995	343.450 294	3.897 730
58 40000	-94397064	5.711 634	308.459 977	4.806 036
59 40000	-88783643	5.903 058	307.416 545	4.341 596
60 40000	-93017025	6.146 512	310.639 975	3.885 067
61 80000	-186834082	10.954 828	1266.378 572	5.054 121
62 80000	-185997521	8.824 858	1228.143 636	4.554 214
63 80000	-182065015	12.113 433	1219.227 569	4.694 382
64 80000	-180803872	10.380 935	1223.968 781	4.738 836
65 100000	-230698391	10.662 057	1965.855 976	6.182 647
66 100000	-230168572	11.152 275	1934.359 841	6.406 684
67 100000	-231393935	14.134 856	1939.487 085	6.116 602
68 100000	-231011693	11.883 698	1921.524 440	6.605 124

Tabella 1: Risultati