Sunday, 23 October, 2022 12:06 PM

Minimum Spanning Tree

→ Prim's
→ Kruskal

Prim's Algorithm

mst set → { 0, 1, 7, 6, 5 }

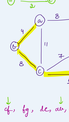
	0	1	2	3	4	5	6	7	8
key	0	4	8	7	14	9	2	1	8

Kruskal's Algorithm

→ grow spanning tree by adding edges
→ greedy approach

1) Sort the edges with respect to weights. (asc)
(ab), (de), (bc), (bd) ...
2) start adding edges to the MST with smallest weight
3) only add edges which does not form cycle

1 + 3 + 5 + 2 = 11



ef, bf, de, ab, df, ef, ce, ab, bc, ad, bd, fg, ac, fg
↓

Prim's → vertices → (graph is dense)

Kruskal → edges

In worst case → Prim's
↓
200 edges

prim/ Kruskal will work ??

3 + 5 = 8

4 + 5 = 10

(4+4)
but work with any weight weight

Union - Find DS

5 people A B C D E

group [A B C D E]
[B C D E]

① B, D, C are connected to each other
② D, E are connected to each other.

* Union Find DS deals with to check whether two people is connected or not in a dynamic way

{A, B, C}, {D, E}

Union (A, B) → connect two elements A & B.

Find (A, B) → check if there is any path from A to B.

Q. S = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }, N=10

arr
↓
to manage connected elements

parent

① Union (1,3)
② Union (1,2)
③ Union (3,4)
④ Union (5,6)
⑤ Union (5,7)
⑥ Union (1,4)
from ⑦ Find (3,5)
then ⑧ Find (4,7)
⑨ Union (2,5)

void SetUnion (arr[], int n) {
for (i=0; i<n; i++) {
arr[i] = i;
}
}

void union (arr, a, b) {
int temp = arr[a];
for (i=0; i<n; i++) {
if (arr[i] == temp)
arr[i] = arr[b];
}
}

void find (arr, a, b) {
if (arr[a] == arr[b])
return true;
else
return false;
}

→ for | union
O(n)
N = 1

Idea: Arr[n] is a parent of A.

S = { 0, 1, 2, 3, 4, 5 }

arr → [0 | 1 | 2 | 3 | 4 | 5]

① Union (1,4)
② Union (3,5)
③ Union (3,4)
④ Union (1,3)
⑤ Union (1,2)
⑥ Find (1,3)
⑦ Find (3,5)

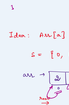
find (0, 5) → a & b
connected if they have the same root.

root (5) → [5 | arr[5]] ...

int root (arr, i) {
while (arr[i] != i)
i = arr[i];
return i;
}

union (arr, a, b) {
root_a = root (arr, a);
root_b = root (arr, b);
arr [root_a] = root_b;
}

bool find (a, b) {
if (root (a) == root (b))
return true;
else
return false;
}



find (3,5)
root (3)
root (5)

(a)

root_a
root_b

log(n)

Union (1,5)

weighted union operation

initialize (arr, n)
for (i=0; i<n; i++)
arr[i] = i;
size[i] = 1;
}

void weighted union (arr, size, a, b) {
root_a = root (a);
root_b = root (b);
if (size [root_a] < size [root_b])
arr[root_a] = arr[root_b];
size [root_b] += size [root_a];
} else {
arr[root_b] = arr[root_a];
size [root_a] += size [root_b];
}
}

S = [0, 1, 2, 3, 4, 5]

size → [1 | 1 | 1 | 1 | 1 | 1]

① Union (0,1)
② Union (1,2) arr = [0 | 0 | 0 | 0 | 4 | 5]
③ Union (3,4)

Union (find → O(log n))

Union with Path Compression

→ worst complexity time root of a tree, we make it as parent to the grandparent (throughout traversing each path, length) whenever I go the root, indirectly cause to children, path while computing the root of A

root (arr, i) {
while (arr[i] != i)
arr[i] = arr[arr[i]];
i = arr [i];
}
return i;
}