Part V: Graph Algorithms

Lecture 12: Kruskal's MST Algorithm

Ahmed Khademzadeh
http://khademzadeh.mshdiau.ac.ir
Azad University of Mashhad

The slides are borrowed from Ke (Kevin) Yi. Thanks to him for his benevolence.

Objective and Outline

Objective: Discuss another algorithm for the MST problem, i.e. Kruskal's algorithm.

Reference: Chapters 21 & 23 of CLRS

- Kruskal's MST algorithm
  - The idea
  - Correctness
  - The algorithm
- The Disjoint Set Union-Find data structure.
  - The basic implementation
  - An improvement

Recalling the Generic Algorithm

- start with an empty graph
- try to add edges one at a time, always making sure that what is built remains acyclic.
- if we are sure every time the resulting graph is always a subset of some minimum spanning tree, we are done.

Lemma

- *$G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *A be a subset of E that is included in some minimum spanning tree for G.*

Let

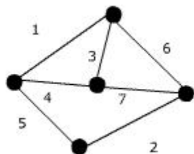
- *$(S, V - S)$ be any cut of G that respects A*
- *(u, v) be a light edge crossing the cut $(S, V - S)$*

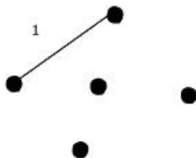
Then, edge (u, v) is safe for A.

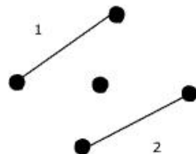
Idea of Kruskal's Algorithm

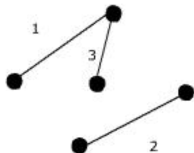
- The **Kruskal's Algorithm** is based directly on the generic algorithm.
- Unlike Prim's algorithm, which grows one tree, Kruskal's algorithm grows a collection of trees (a forest).
- Initially, trees of the forest are the vertices (no edges).
- In each step add the cheapest edge that does not create a cycle.
- Continue until the forest 'merge to' a single tree.

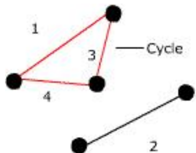
Example

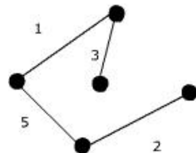












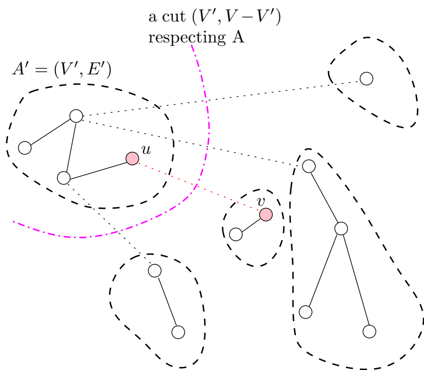
Outline

- Kruskal's MST algorithm
  - The idea
  - <u>Correctness</u>
  - The algorithm
- The Disjoint Set Union-Find data structure.
  - The basic implementation
  - An improvement

Why Kruskal's Algorithm is Correct?

Let A: edge set selected by Kruskal's Algorithm; (u, v): edge to be added next. It suffices to show

1. there is a cut which respects A, and
2. (u, v) is the light edge crossing the cut.



- Let $A' = (V', E')$ denote the tree of the forest A that contains u. Consider the cut $(V', V - V')$.
- There is no edge in A crosses this cut, so the cut respects A.
- Since adding (u, v) to A' does not induce a cycle, (u, v) crosses the cut.
- Moreover, since (u, v) is currently the smallest edge, (u, v) is the light edge crossing the cut.

Outline

- Kruskal's MST algorithm
  - The idea
  - Correctness
  - The algorithm
- The Disjoint Set Union-Find data structure.
  - The basic implementation
  - An improvement

Outline of Kruskal's Algorithm

1. Set $A = \emptyset$ and $F = E$, the set of all edges.
2. Choose an edge e in F of minimum weight, and check whether adding e to A creates a cycle.
   - If "yes", remove e from F.
   - If "no", move e from F to A.
3. If $F = \emptyset$, stop and output the minimal spanning tree (V, A). Otherwise go to Step 2.

Questions

- How does algorithm choose edge $e \in F$ with minimum weight?
- How does algorithm check whether adding e to A creates a cycle?

How to Choose the Edge of Least Weight?

Question

How does algorithm choose edge $e \in F$ with minimum weight?

Answer:

- Start by sorting edges in E in order of increasing weight.
- Walk through the edges in this order.
- (Once edge e causes a cycle it will always cause a cycle so it can be thrown away.)

How to Check for Cycles?

Observations:

- At each step of the outlined algorithm, (V, A) is acyclic so it is a forest.
- If u and v are in the same tree, then adding edge $\{u, v\}$ to A creates a cycle.
- If u and v are not in the same tree, then adding edge $\{u, v\}$ to A does not create a cycle.

Question

How to test whether u and v are in the same tree?

High-Level Answer: Use a disjoint-set data structure

- Vertices in a tree are considered to be in same set.
- Test if Find-Set(u) = Find-Set(v)?

Low-Level Answer:

- The UNION-FIND data structure implements this

The UNION-FIND Data Structure

UNION-FIND supports three operations on collections of disjoint sets. Let n be the size of the universe.

1. Create-Set(u): Create a set containing the single element u.
   - $O(1)$
2. Find-Set(u): Find the set containing the element u. (Say each set has a unique ID)
   - $O(\log n)$
3. Union(u, v): Merge the sets containing u and v respectively into a common set.
   - $O(\log n)$

For now we treat UNION-FIND as a black box. Will see implementation later.

Kruskal's Algorithm: Details

```
Sort E in increasing order by weight w; // O(|E| log |E|)
// After sorting E = ⟨{u_1, v_1}, {u_2, v_2}, ..., {u_|E|, v_|E|}⟩
A = { };
foreach u in V do CREATE-SET(u) // O(|V|)
for e_i = (u_i, v_i) from 1 to |E| do
    // O(|E| log |V|)
    if FIND-SET(u_i) ≠ FIND-SET(v_i) then
        add {u_i, v_i} to A;
        UNION(u_i, v_i);
    end
end
return A;
```

Remark: With a proper implementation of UNION-FIND, Kruskal's algorithm has running time $O(|E| \log |E|) = O(|E| \log |V|)$.

Outline

- Kruskal's MST algorithm
  - The idea
  - The algorithm
  - Correctness
- <u>The Disjoint Set Union-Find data structure</u>
  - The basic implementation
  - An improvement

Disjoint Set Union-Find

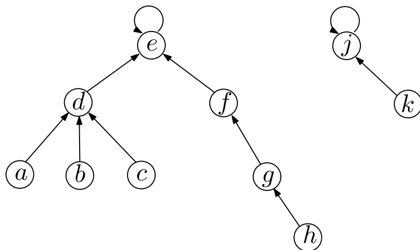
A disjoint set Union-Find date structure supports three operation on x and y:

1. Create-Set(x)
   - Create a set containing a single item x.
2. Find-Set(x)
   - Find the set that contains x
3. Union(x, y)
   - Merge the set containing x, and another set containing y to a single set.
   - After this operation, we have Find-Set(x) $=$ Find-Set(y).

Outline

- Kruskal's MST algorithm
  - The idea
  - The algorithm
  - Correctness
- The Disjoint Set Union-Find data structure
  - The basic implementation
  - An improvement

Up-Tree Implementation



- Every item is in a tree.
- The root of the tree is the representative item of all items in that tree
  - i.e., the root of the tree represents the whole items.
  - use the root's ID as the unique ID of the set.
- In this up-tree implementation, every node (except the root) has a pointer pointing to its parent.
  - The root element has a pointer pointing to itself.

Create-Set(x) and Find-Set(x)

Create-Set(x): easy

```
x→parent=x;
```

Find-Set(x): also easy

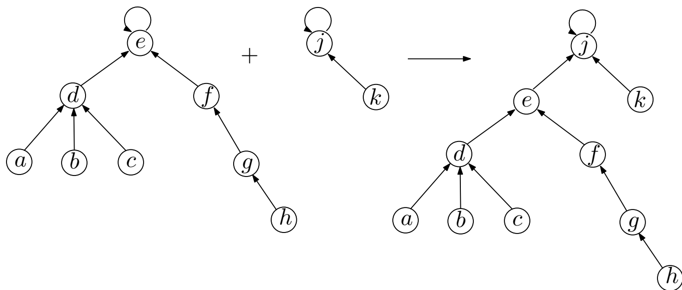
- simply trace the parent point until we hit the root, then return the root element.

```
while x != x→parent do
|   x = x→parent;
end
return x
```

Union(x, y)

Naive solution:

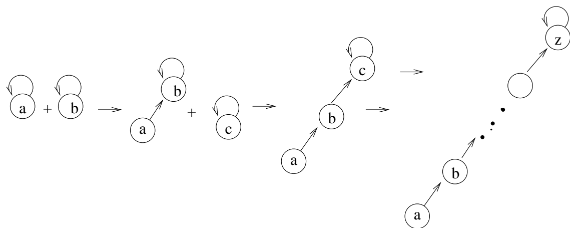
- put the parent pointer of the representation of x pointing to the representation of y.



Question

Is it a good idea?

Problem



May become a linked-list at the end! Hence it is not efficient.

Question

Can we do better?

Simple trick (Union by height):

- when we union two trees together, we always make the root of taller tree the parent of shorter tree.

Up-Tree Implementation : Union by Height

- The root of every tree also holds the height of the tree.
- In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

Union(x, y)

```
a=Find-Set(x);
b=Find-Set(y);
if a.height ≤ b.height then
    if a.height == b.height then
        b.height++;
    end
    a→parent=b;
else
    b→parent=a;
end
```

Lemma

For the root x of any tree, let $size(x)$ be the number of nodes and $h(x)$ be the height of the tree. We have $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before $\text{Union}(x, y)$. Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have

     $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}.$$

   - $h(x) = h(y)$, we have

     $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} = 2^{h(y)+1} = 2^{h(x')}.$$

   - $h(x) > h(y)$, it is similar to the first case

Lemma

For n items, the running time of

- *Create-Set is $O(1)$,*
- *Find-Set is $O(\log n)$, and*
- *Union is $O(\log n)$*

respectively.

Proof.

- Obviously, Create-Set(x) is $O(1)$, and the running time of Union(x, y) depends on Find-Set(x).
- Since the running time of Find-Set(x) depends on the height of the tree. From previous lemma, for any tree, we have

$$\begin{aligned} n \geq 2^h \quad &\Rightarrow \quad h \leq \log n \\ &\Rightarrow \quad h = O(\log n) \end{aligned}$$

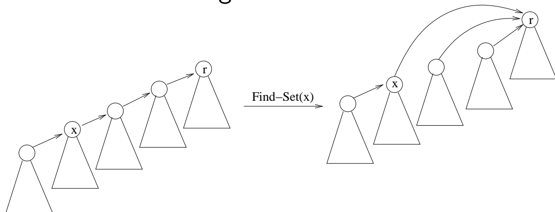
  Hence we have Find-Set(x) $= O(\log n)$.

Outline

- Kruskal's MST algorithm
  - The idea
  - The algorithm
  - Correctness
- The Disjoint Set Union-Find data structure
  - The basic implementation
  - An improvement

Up-Tree Implementation: Path Compression

- We can make the running time even faster if we add another trick.
- In Find-Set(x), we trace the path from x to the root.
- Let r be the root of the tree, and the path from x to r is $x a_1 a_2 \ldots a_k r$.
- As a by-product, we also make all the parent pointers of $x, a_1, a_2, \ldots a_k$ pointing to r directly.
  - Shortens the time of some future calls to Find-Set.
  - Does not increase height.



- This idea is called path compression.

Path Compression...

Question

Does path compression improves the running time of union-find?

$\lg^{(i)} n$: defined recursively for nonnegative integers i as

$$\lg^{(i)} n = \begin{cases} n & \text{if } i = 0 \\ \lg(\lg^{(i-1)} n) & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0, \\ \text{undefined} & \text{if } i > 0 \text{ and } \lg^{(i-1)} n \leq 0, \text{ or } \lg^{(i-1)} n \text{ is undefined.} \end{cases}$$

The iterated logarithm is defined as

$$\lg^* n = \min \{i \geq 0 : \lg^{(i)} n \leq 1\}$$

- a very slow growing function.
- e.g.,
  $\lg^* 2 = 1, \lg^* 4 = 2, \lg^* 16 = 3, \lg^* 65536 = 4, \lg^* 2^{65536} = 5.$

Path Compression...

The following theorem is stated without proof.

Theorem

A sequence of m Create-Set, Find-Set and Union operations, n of which are Create-Set operations, can be performed on a disjointed-set forest with union by height and path compression in worst-case time $O(m \lg^* n)$.

Question

What is the running time of Kruskal's algorithm if we employ this implementation of disjoint set Union-Find?