PhD Foundation: Minimum Spanning Tree

ZHANG Shiqi

July 29, 2019

1 Basic Definition

Cut: A cut is a partition of the vertices of a graph into two disjoint subsets. A cut $C = (S, T)$ is a partition of V of a graph $G = (V, E)$ into two subsets S and T. The cut-set of a cut $C = (S, T)$ is the set $\{(u, v) \in E \mid u \in S, v \in T\}$ of edges that have one endpoint in S and the other endpoint in T.

Light Edge: An edge crossing a cut is called **light** if it has the smallest weight of any edge crossing the cut.

Respect: We say a cut **respects** edge set A if no edges in A cross the cut.

Minimum Spanning Tree: MST is a subset of an undirected graph $G(V, E, W)$. In the MST, there is no cycle and the total weights are minimum.

Safe Choice: Let A be a set of edges, and consider a cut that respects A. Suppose there is an MST containing A. Let (u, v) be a light edge. Then there is an MST containing $A \cup (u, v)$. (Proof refers to slides [1])

2 Disjoint-Set Data Structure

Definition: Disjoint-set data structure is a data structure representing a dynamic collection of sets $\{S_1.S_2, ..., S_n\}$. It can keep track of the changing for connected components.

Functions: In each connected component, we need to find a unique element to identify and represent current component. This unique element is called the representative of component S, denoted by $rep[S]$. In order to keep track of the connected components, a disjoint-set data structure needs to contain the following three functions:

1. MakeSet(u): create a new set containing u.
2. Find(u): find the representative of the set containing u.
3. Union(u,v): merge two sets containing u and v respectively. Update the representative of new set.

[1] `http://web.stanford.edu/class/archive/cs/cs161/cs161.1176/Slides/Lecture15.pdf`

2.1 Implemented by Linked List

Each set can be represent by the data structure in Figure 1. We use the first element in the linked list as the representative, and it is directly pointed by *head*. In order to eliminate redundant walking, every element use a traceback pointer to point back to the set, and the *tail* points to the last element. The advantages will be illustrated in the following.

Functions:

1. MakeSet(u): point both *head* and *tail* to u, and point u back to the set. **O(1)**.
2. Find(u): use the traceback pointer of u to find the set and *head* inside, the representative is the next of *head*. **O(1)**. (Without the traceback pointer, we need to use double linked list and walk from u to *head*, the worst case is **O(n)**.)
3. Union(u,v): redirect the trackback pointers of elements in second set to first set, and link the last element in first set to the first element in the second set and point the tail to the last element in the second set. **O(k)**, where k is the size of second set. (If implemented by double linked list, it is **O(n)**.)

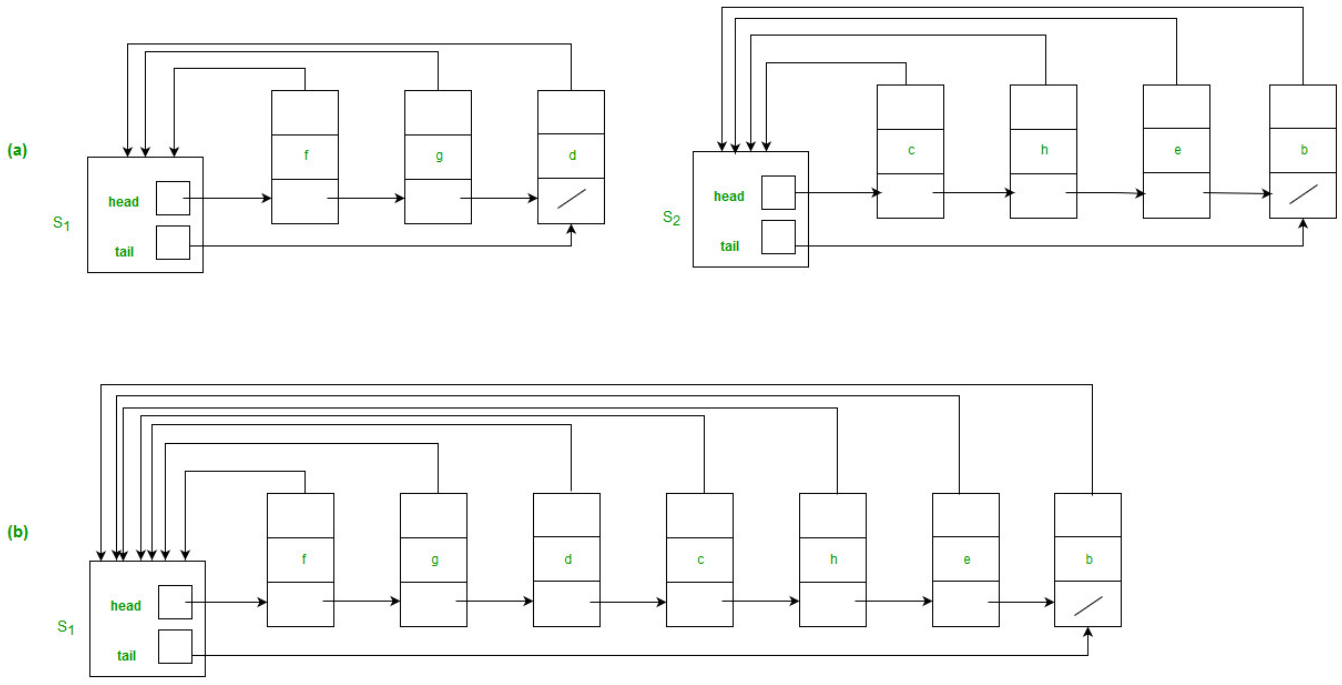

Figure 1: Linked List Disjoint Set Data Structure

Optimization: Heuristic Weighted Strategy: In the structure of set $\{head, tail\}$, add an attribute *weight* to record the elements inside, i.e $\{head, tail, weight\}$. During the union operation, the set with smaller weight will merge to that with larger weight. For a element u and the initial set S_u, if u updates its pointer, then the size of S_u will at least be doubled. Then for a set with k elements, the union time for each element inside is $O(\log k)$. Thus, for n elements, the total time for union operation is $O(n \log n)$.

2.2 Implemented by Tree

We use the unique root node in a tree to be the representative of the component. We store the tree into an array called *parent*. In the array, the index represents the node itself and the element of the given index means the parent node of it.

Functions:

1. MakeSet(u): initialize a tree with root u by setting $parent[u] = u$. **O(1)**.
2. Find(u): walk uo from u to the root. **O(log n)**.
3. Union(u,v): set $rep[v]$ to $rep[u]$ by setting $parent[rep[v]] = parent[rep[u]]$. **O(log n)**.

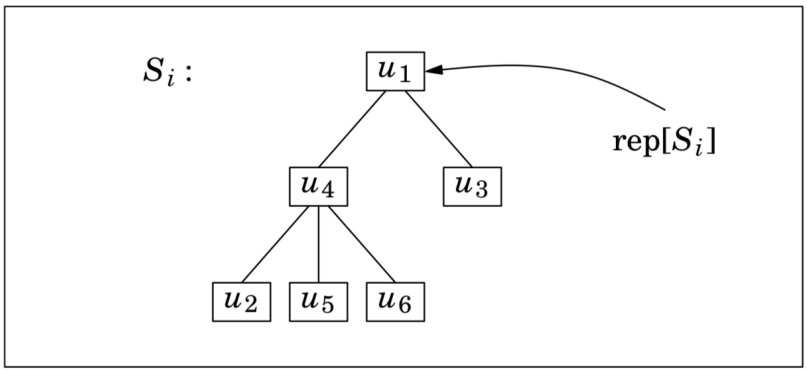

Figure 2: Forest of Tree Disjoint Set Data Structure

Optimization: Union by Rank: Like heuristic weight strategy in linked list, an attribute *rank* will be added to represent the height of a tree. The tree with less rank will be added into the tree with larger rank.

Optimization: Path Compression: Path compression aims at decrease the height of tree and creates a flatten tree. When invoking $Find(u)$, it will link all nodes from u to root into the root.

3 Algorithm

3.1 Kruskal's Algorithm

The basic idea of Kruskal's Algorithm is to dynamic merge two connected components with the least weight edge according to disjoint-set data structure.

1. Initially, in graph $G(V, E, W)$, every vertex is split into a single connected component. Every edge is sorted ascending by weight.
2. During each iteration, let's try to traverse the sorted edges. If next least weight edge can connect two connected components, then merge two components by this edge. If not, skip it.
3. Stop it when only one connected component exists. (Or tried every edge)

3.2 Prim's Algorithm

Prim's algorithm only maintains one connected component S and choose the edge in cut $(S, V-S)$ with minimum weight. The implementation is quite like Dijkstra Algorithm and we can use priority queue to speed it up.

1. Random pick a vertex as the starting point of MST and initialize a heap. In the heap, the element is the vertex and the key is the weight of edge connecting to the vertex.
2. During the iteration, select the top element of heap (minimum weight edge) as current source node. Update the key of its neighbors if current weight is smaller and the neighbor is not in MST and add the neighbors into heap.
3. Stop until the heap is empty.

3.3 Why can Prim's and Kruskal's Algorithm get Minimum Spanning Tree?

Proof. Let's prove it by contradiction. Suppose the tree S generated by Kruskal's or Prim's algorithm are not the minimum one, then there exists a MST T.

Suppose e is the least weighted edge in $S-T$. If we add edge e into tree T, i.e, $T \cup \{e\}$, there exist a cycle inside and we need to remove a edge f of T. The new tree $T' = (T \cup \{e\}) \backslash f$. Since T is MST, then $w(f) \leq w(e)$. However, Kruskal and Prim algorithm select the edge with minimum weight, i.e $w(e) \leq w(q)$. Thus, $w(f) = w(e)$ and $S = T'$ is a MST. □

3.4 Time Complexity

Kruskal's Algorithm: The time complexities for both $Find()$ and $Union()$ are $O(\log n)$. The complexity for $makeSet()$ is $O(1)$.

1. Initialization: $V * makeSet() + Sort(E) = V + O(E \log E)$
2. Loop: $2O(E) * Find() + O(E) * Union() = 3O(E \log V)$
3. Total: $O(E \log V)$ since $E \leq V^2$

Prim's Algorithm: The total time complexity is $BuildHeap(V) + V * ExtractMin() + O(E) * DecreaseKey()$. If we use binary heap with $BuildHeap() = O(n)$, $ExtractMin() = O(\log n)$ and $DecreaseKey() = O(\log n)$, the total time complexity is $O(E \log V)$.