AI2615 算法设计与分析 2020-2021 春季学期

Lecture 6 – Greedy Algorithms: Kruskal's Algorithm

2021 年 4 月 2 日

Lecturer: 张驰豪 *Scribe:* 陶表帅

Greedy algorithms are a type of iterative algorithms that myopically choose the locally optimal choice at each iteration. For example, when a student is facing many homework problem sets that have different deadlines of submission, a typical greedy algorithm always iteratively chooses a homework problem set with the earliest deadline. As another example, in a board game, a greedy algorithm will always play a locally best move, without further consideration of the future. For some problems, the greedy algorithm is provably optimal.

Problem 1 (Minimum Spanning Tree (MST)). Given a weighted undirected graph $G = (V, E, w)$ such that $w(e) > 0$ for any $e \in E$, find a connected subgraph $T = (V, E')$ containing all vertices of G with the minimum weight:

$$w(T) \triangleq \sum_{e \in E'} w(e).$$

A subgraph T with minimum weight is called a *minimum spanning tree*.

First of all, it is easy to check a minimum spanning tree T of a graph G is indeed a tree. Suppose otherwise. There must be a cycle in T. Since $w(e) > 0$ for all $e \in E$, removing any edge of T can reduce the weight of T while still guaranteeing that T is connected.

Notice that T must contain all vertices of the original graph G. The set of edges E' is sufficient to describe T. We will describe a minimum spanning tree by a set of edges, instead of a subgraph. Notice that a minimum spanning tree T contains exactly $n - 1$ edges: $|T| = n - 1$.

Figure 1 presents an example of a graph with its minimum spanning tree.

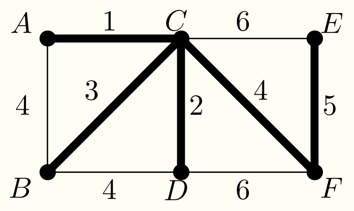

Figure 1: A graph with its minimum spanning tree. Bold edges form a minimum spanning tree.

1 Kruskal's Algorithm

Kruskal's algorithm is a typical greedy algorithm that finds a minimum spanning tree. Kruskal's algorithm can be described simply as follows. Starting from setting $T = \emptyset$, it iteratively adds an edge with minimum weight while making sure T do not have any cycle, and it terminates when T contains exactly $n-1$ edges. Using the graph in Figure 1 as an example, Kruskal's algorithm will iteratively choose the edges $\{A,C\},\{C,D\},\{B,C\},\{C,F\}$ and $\{E,F\}$.

1.1 Correctness of Kruskal's Algorithm

We first prove the following proposition.

Proposition 2. *Given a weighted undirected graph $G = (V, E, w)$ with positively-weighted edges, consider an arbitrary subset of edges $X \subseteq E$ such that X is a subset of a minimum spanning tree. For any $S \subsetneq V$ such that $X \cap E(S, V \setminus S) = \emptyset$,*[1] *$X \cup \{e^*\}$ is a subset of a minimum spanning tree, where $e^* \in \operatorname{argmin}_{e \in E(S,V\setminus S)} w(e)$*[2] *is an edge in $E(S, V \setminus S)$ with minimum weight.*

Proof. We prove it by contradiction. Suppose no minimum spanning tree contains $X \cup \{e^*\}$. Let $e^* = \{u, v\}$ where $u \in S$ and $v \in V \setminus S$. Let T be a minimum spanning tree containing X. By our assumption, $e^* \notin T$. Since T is connected, there exists a path connecting u and v in T. This path, together with $e^* = \{u, v\}$, form a cycle. Since $u \in S$ and $v \in V \setminus S$, there exists an edge $e' = \{u', v'\} \in T$ such that $u' \in S$ and $v' \in V \setminus S$. Consider the subgraph with edge set $T' = T \cup \{e^*\} \setminus \{e'\}$. This subgraph is still connected: we know $T \cup \{e^*\}$ contains a cycle and e' is on the cycle.
As $e^* \in \operatorname{argmin}_{e \in E(S,V\setminus S)} w(e)$ and $e' \in E(S, V \setminus S)$, we have $w(e') \geq w(e)$. If $w(e') > w(e)$, T' has less weight than T, which contradicts to that T is a minimum spanning tree. If $w(e') = w(e)$, the weights of T' and T is equal. Thus, T' is also a minimum spanning tree. We have $X \subseteq T'$, as the only edge removed from T is $e' \in E(S, V \setminus S)$ and $X \cap E(S, V \setminus S) = \emptyset$. We also have $e^* \in T'$ by our construction. Therefore, $X \cup \{e^*\} \subseteq T'$, which contradicts to our assumption that no minimum spanning tree contains $X \cup \{e^*\}$. □

Theorem 3. *Kruskal's algorithm always finds a minimum spanning tree.*

Proof. Let X_i be the set of edges after i-th round of the algorithm. It suffices to show that X_i is always a subset of a minimum spanning tree for any i. We prove this by induction on i.
The base step is trivial: $X_0 = \emptyset$ is a subset of any minimum spanning tree.
For the inductive step, suppose X_i is a subset of a minimum spanning tree. Let $e^* = \{u, v\}$ be the edge added to T in the next iteration. Let $S_u \subseteq V$ be the set of all vertices reachable from u by edges in X_i. We know

[1] $E(S, V \setminus S)$ is the set of edges (u, v) such that $u \in S$ and $v \in V \setminus S$.

[2] $\operatorname{argmin}_{e \in E(S,V\setminus S)} w(e)$ is the set of edges in $E(S, V \setminus S)$ that have minimum value of $w(e)$. Notice that there may be more than one edge with minimum weight.

$v \notin T$, for otherwise adding e^* to T will create a cycle implying that e^* should not have been added by the algorithm. Therefore, $e^* \in E(S_u, V \setminus S_u)$. Moreover, we have $X_i \cap E(S_u, V \setminus S_u) = \emptyset$, as S_u already contains all vertices reachable from u by edges in X_i. This also implies any edge in $E(S_u, V \setminus S_u)$ can potentially be chosen by the algorithm. Since the algorithm always chooses a valid edge with minimum weight, we have $e^* \in \operatorname{argmin}_{e \in E(S_u, V \setminus S_u)} w(e)$. By Proposition 2, $X_{i+1} = X_i \cup \{e^*\}$ is contained in a minimum spanning tree. □

1.2 Implementation of Kruskal's Algorithm

To implement Kruskal's algorithm, we need a data structure that stores subsets of vertices, where each subset contains vertices that are mutually connected by currently selected edges. For each subset, we define a vertex as its representative. We need following subroutines.

- MakeSet(u): create a set containing a single vertex u.
- Find(u): find the representative of the set containing u.
- Union(u, v): merge the set containing u and the set containing v.

The implementations of these three subroutines are deferred to the next section. Given these three subroutines, Kruskal's algorithm is described in Algorithm 1.

Algorithm 1 Kruskal's algorithm

```
Kruskal($G = (V, E, w)$)
1: for each $u \in V$, MakeSet($u$)
2: $T \leftarrow \emptyset$
3: sort $E$ by ascending order of $w(\cdot)$
4: for each $e = \{u, v\} \in E$ in the order above:
5:     if Find($u$) $\neq$ Find($v$):
6:         $T \leftarrow T \cup \{e\}$
7:         Union($u, v$)
8:     endif
9: endfor
10: return $T$
```

2 Union-Find Set

In this section, we discuss how to implement the three subroutines, MakeSet, Find, and Union. This can be implemented using a data structure called the *union-find set*. In a union-find set, each set is represented by a tree, where nodes of the tree are elements of the set. The root node of the tree is the representative of the corresponding set.

Let $\pi(u)$ denote the parent of u. Let $\pi(u) = u$ if u is a root. Let $\text{rank}(u)$ be the height of the subtree rooted at u. The implementations of the three subroutines are as follows.

MakeSet(u):

```
• $\pi(u) \leftarrow u$;
• $\text{rank}(u) \leftarrow 0$;
```

Find(u):

```
• while $\pi(u) \neq u$: $u \leftarrow \pi(u)$;
• return $u$;
```

Union(u, v):

```
• $r_u \leftarrow$ Find($u$) and $r_v \leftarrow$ Find($v$);
• if $r_u = r_v$: return;
• if $\text{rank}(r_u) > \text{rank}(r_v)$: $\pi(r_v) \leftarrow r_u$;
• else: $\pi(r_u) \leftarrow r_v$;
•     if $\text{rank}(r_u) = \text{rank}(r_v)$: $\text{rank}(r_v) \leftarrow \text{rank}(r_v) + 1$;
```

The implementations of MakeSet and Find are straightforward. Union(u, v) first find the roots of the two trees containing u and v respectively. It then attaches the root of the tree with the lower height as a child of the root of the tree with the higher height. Figure 2 gives an example of a sequence of Union operations.

2.1 Time Complexity

The time complexity for MakeSet(u) is clearly $O(1)$.

The worst case time complexity of Find(u) is the height of the tree when u is a deepest leaf node, as we need to travel along the path from u to the root. The time complexity for Find(u) is the height of the tree (asymptotically). In Union(u, v), we always merge two trees such that the tree with lower height is attached to the tree with a higher height. This significantly reduces the height of the tree.

Proposition 4. *A subtree rooted at a node of rank k contains at least 2^k nodes.*

Proof. We prove this by induction.

For the base step, the subtree rooted at a node with rank 0 only contains one node (itself), and we have $2^0 = 1$.

For the inductive step, suppose for any node with rank k, the subtree rooted at it contains at least 2^k nodes. By our definition of subroutine Union, whenever a node's rank become $k+1$, it must be that two trees

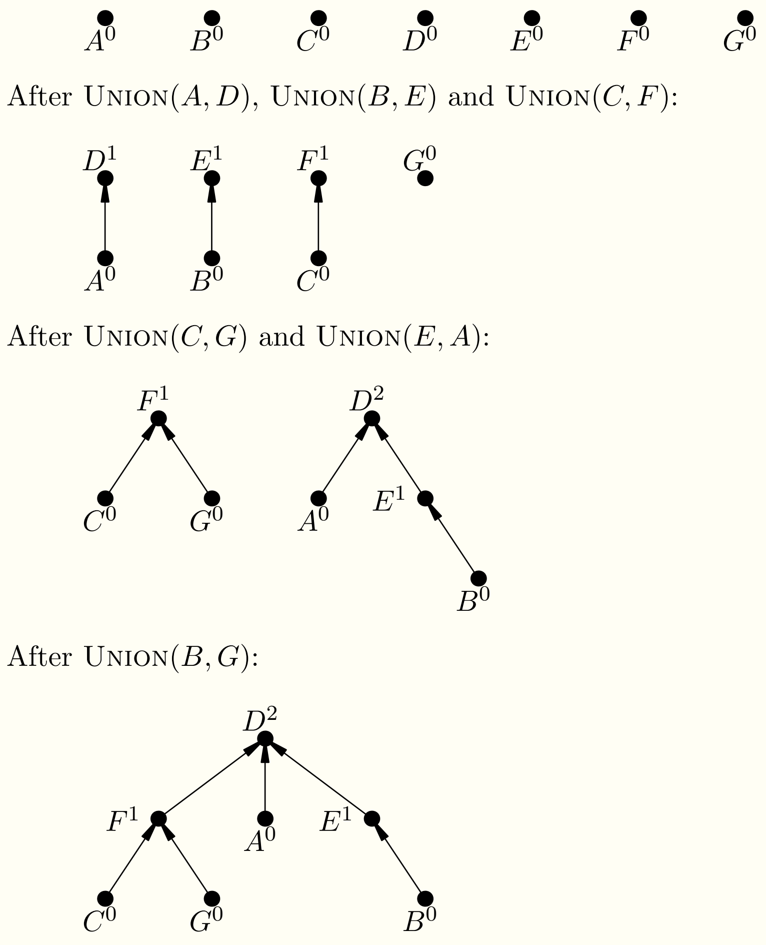

Figure 2: An example of a sequence of Union operations. Superscripts denote the ranks of the nodes.

whose roots have rank k have been merged. By induction hypothesis, each of the two tree contains at least 2^k nodes. Thus, the merged tree contains at least $2^k + 2^k = 2^{k+1}$ nodes. This concludes the inductive step. □

Proposition 5. *Once a node become an internal node (i.e., not a root), its rank remains the same ever after.*

Proof. By our definition of Union, only root node has a chance to change its rank. □

Proposition 6. *Every parent has a strictly higher rank than the rank of its child. That is,* $\text{rank}(\pi(x)) > \text{rank}(x)$ *whenever x is not a root.*

Proof. This again follows from our definition of Union. Initially, all nodes are isolated. Whenever two trees are merged, the root with lower rank becomes a child of the root with higher rank, so $\text{rank}(\pi(x)) > \text{rank}(x)$ is preserved. If two roots have the same rank, the new root's rank is increased by 1 after merging, which again preserves $\text{rank}(\pi(x)) > \text{rank}(x)$. □

As a corollary to the proposition above, if two nodes u, v satisfy $\text{rank}(u) = \text{rank}(v)$, then the subtree rooted at u is disjoint to the subtree rooted at v. If this is not the case, then either u is an ancestor of v or v is an ancestor of u, and Proposition 6 implies that this is impossible.

Proposition 7. *The number of the nodes with rank k is at most $n/2^k$.*

Proof. Let n_k be the number of of nodes with rank k. By Proposition 4, each of those n_k nodes is a root of a subtree with at least 2^k nodes. By the corollary to Proposition 6, these subtrees are disjoint. Therefore, we have identified at least $n_k \cdot 2^k$ nodes. Thus, $n \geq n_k \cdot 2^k$, which implies $n_k \leq n/2^k$. □

Proposition 7 implies that the maximum rank over all nodes is at most $\log_2 n$. This means that the maximum depth of the tree is at most $\log_2 n$. Therefore, the time complexity of $\textsc{Find}(u)$ is $O(\log n)$.
Finally, $\textsc{Union}(u, v)$ only requires two executions of $\textsc{Find}$ and some operations that can be done in a constant time. The time complexity of $\textsc{Union}(u, v)$ is $O(\log n)$.
Let us turn back to Kruskal's algorithm. The initialization of the n sets requires $O(n)$ time. Sorting all edges requires $O(m \log m) = O(m \log n)$ time. For the main part, each edge is only processed once, with a constant number of $\textsc{Find}$ and $\textsc{Union}$ operations. The time complexity for the main part is therefore $O(m \log n)$. Putting together, the time complexity of Kruskal's algorithm is $O(m \log n)$.

3 Refining Kruskal's Algorithm by Path Compression

The bottleneck for the time complexity of Kruskal's algorithm is the sorting step and the main part, each of which require $O(m \log n)$ time. In some applications, the edges of a weighted graph have already been sorted beforehand. In some other applications, the weights of those edges are upper-bounded by a small number, in which case sorting only requires $O(m)$ time. In either case, the main part of Kruskal's algorithm is the only bottleneck. In this section, we will see that the time complexity of the main part of Kruskal's algorithm can be further reduced by some refined implementations of the subroutine $\textsc{Find}$.

Path compression The refinement is simple: whenever we implement $\textsc{Find}(u)$, after we have searched the root of the tree containing u, we attach u as a direct child of the root. This operation does not affect the correctness of the algorithm, as all we care is whether u is contain in the set of vertices represented by the tree. On the other hand, this operation can potentially reduced the height of the tree. The new implementation of $\textsc{Find}(u)$ is shown below.

```
FIND($u$):
    • if $u \neq \pi(u)$: $\pi(u) \leftarrow$ FIND($\pi(u)$);
    • return $\pi(u)$;
```

Notice that we choose not to update the rank of the nodes during Find(u). As a result, since the new implementation can reduce the height of the tree, the rank of a node may no longer represent the height of the subtree rooted at this node.

Nevertheless, most of the propositions in the previous section still hold. Proposition 5, 6 and 7 still hold here. In particular, the proofs for Proposition 5 and 6 remains unchanged. For Proposition 7, it suffices to notice that the new implementation of Find only changes the way nodes are attached, not the rank of any node. Therefore, if Proposition 7 holds before, Proposition 7 also holds here.

Figure 3 illustrates an example of a sequence of Find operations with the new path compression technique introduced above.

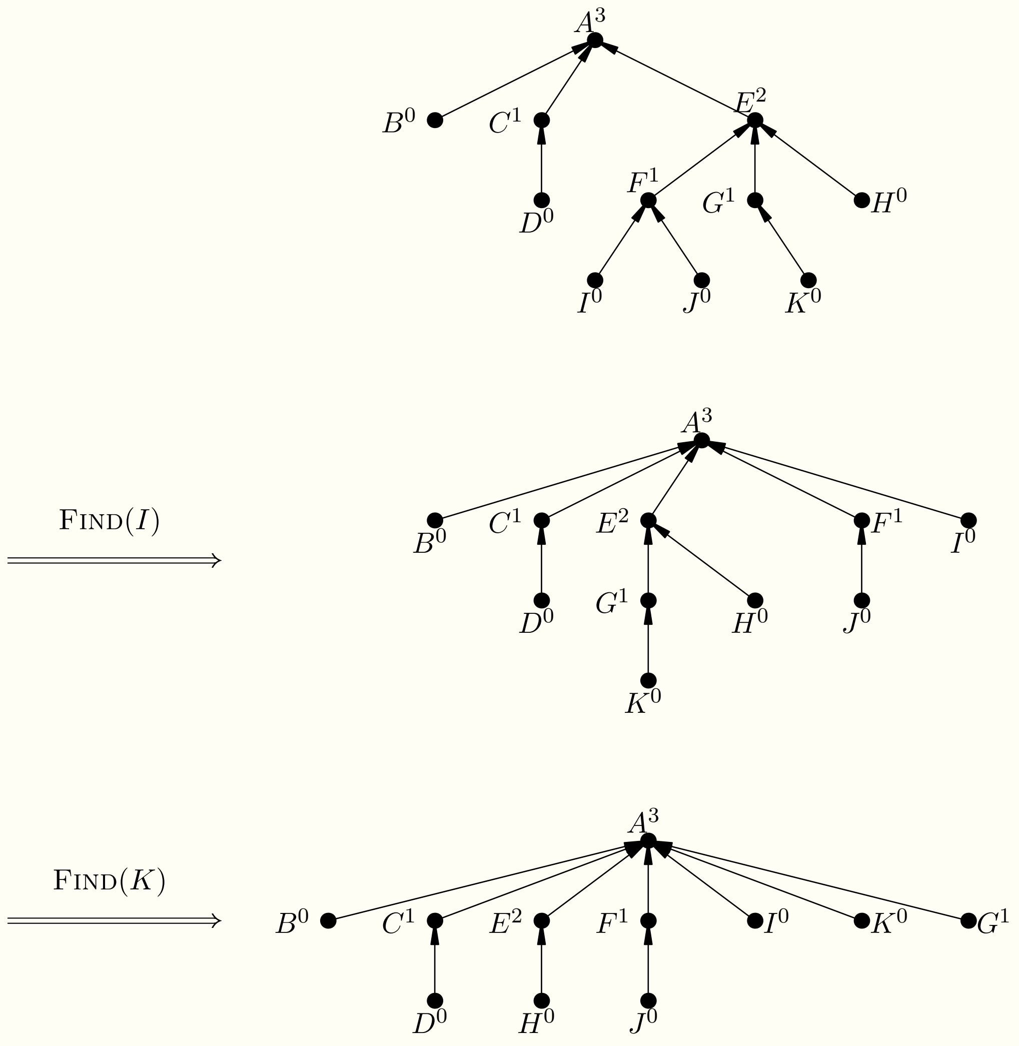

Figure 3: An example of a sequence of Find operations with path compression. Superscripts denote the ranks of the nodes.

3.1 Time Complexity

What is the time complexity for the main part of Kruskal's algorithm with this path compression refinement? Tarjan [1975] showed that the time complexity is $O(m\alpha(n))$, where $\alpha(n)$ is the inverse of the Ackerman function. Although $\alpha(n)$ is unbounded, it grows extremely slowly. To see how slow it is, if N represents the number of atoms in our universe, we have $\alpha(N) \leq 4$; we have $\alpha(9876!) = 5$. We can treat $\alpha(n)$ as a constant in any real life application. We say that the *amortized* time complexity for a Find operation takes $O(\alpha(n))$ time, meaning that the average time for a Find operation is $O(\alpha(n))$. Recall that we have introduced the concept of amortized time complexity when we were introducing Fibonacci heap in the last lecture.

In this section, we show a weaker result that the running time is $O(m\log^* n)$, where $\log^* n$ is the number of log operations required to make n less than 1. That is, $\log^* n$ equals to k where

$$\underbrace{\log\log\cdots\log}_{k \text{ logs}} n \leq 1.$$

$\log^* n$ still grows slowly enough such that it can be treated as a constant in any real life applications. For example, we have

$$\log^*\left(2^{2^{2^{2^2}}}\right) = \log^*\left(2^{65536}\right) = 5.$$

This result is due to Hopcroft and Ullman [1973].

Firstly, we partition $\{1,\ldots,n\}$ to $\log^* n$ levels as follows:

$$\{1\},\{2\},\{3,4\},\{5,6,\ldots,16\},\{17,\ldots,2^{16}\},\cdots,\{k+1,\ldots,2^k\},\{2^k+1,\ldots,2^{2^k}\},\cdots$$

Next, we will use a clever *charging argument* to analyze the time complexity. This works as follows. We will award each node certain amount of money, such that the total money awarded is $O(n\log^* n)$ dollars. We will then define a special kind of Find operations where a node need to spend \$1 to execute. Next, we will show that each node's received money is sufficient to pay for all this kind of operations throughout the algorithm. Finally, we will show that each execution of Find(u) requires a number of steps that is equal to $O(\log^* n)$ plus the money paid by the nodes on the path from u to its root relevant to this Find(u) operation. Therefore, the overall time complexity is given by $O(m\log^* n)$ plus the total money all nodes have paid during the algorithm. Since each node's received money is sufficient to pay for its cost, the overall time complexity is $O(m\log^* n) + O(n\log^* n) = O(m\log^* n)$.

Defining each node's award and payment Whenever a root node u becomes an internal node during the algorithm, we award it 2^k dollars if $\text{rank}(u)$ is in the level $\{k+1,\ldots,2^k\}$. Proposition 5 implies that the rank of u will be fixed from now on. It is also clear that u will no longer become a root again.

Whenever Find(u) is executed, we charge \$1 from node u if $\text{rank}(\pi(u))$ and $\text{rank}(u)$ are in the same level and $\pi(u)$ is not a root. Notice that there are many recursive Find operations in Find(u). Node u is only charged \$1 once during FIND(u), and those other FIND operations in FIND(u) are charged from u's ancestors when applicable (i.e., when any v of u's ancestors satisfies that $\text{rank}(\pi(v))$ and $\text{rank}(v)$ are in the same level and $\pi(v)$ is not a root).

Now let us calculate the total amount of money awarded to the nodes. By the end of the algorithm, Proposition 7 says that the number of nodes with rank k is at most $n/2^k$. Thus, the number of nodes with rank strictly larger than k is at most $\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}$. Since we have argued that the ranks of the nodes will be fixed once they receive awards and those nodes will never be roots again (which means they will not be awarded more than once), the total amount of money paid to those node is at most $2^k \cdot \frac{n}{2^k} = n$ dollars. In particular, those nodes with ranks fall into the level $\{k+1, \ldots, 2^k\}$ is a subset of the set of nodes with ranks strictly larger than k, and we need to pay at most n dollars to them. The total amount of money awarded to all nodes is therefore $O(n \log^* n)$, as there are $\log^* n$ levels.

Proving that each node's award is sufficient to cover the cost it needs to spend A crucial observation here is that, whenever \$1 is charged from a node u, u's new parent must have a higher rank than that of its old parent. This is because, after FIND(u), u is attached to the root, which is an ancestor of u's current parent, and Proposition 6 implies the root has a higher rank. The only case when the rank of u's parent is no smaller than the rank of the root is that u is a root itself or u is the child of a root, and our rule indicates that u will not be charged in this case. With this observation, any node whose rank is in the level $\{k+1, \ldots, 2^k\}$ will be charged no more than $2^k - k < 2^k$ dollars, which is less than the award it receives.

We remark that once $\text{rank}(\pi(u))$ and $\text{rank}(u)$ are in different levels. They will remain in different levels in the remaining part of the algorithm, even if FIND(u) may be called again and $\pi(u)$ may be changed after that. By our above observation and Proposition 5, the rank of $\pi(u)$ can only change from a level higher than the level of $\text{rank}(u)$ to a level that is even higher, and $\text{rank}(u)$ itself will never change.

As another remark, our claim that each node's award is sufficient holds for an arbitrarily number of FIND operations. Intuitively, when all nodes are called with the FIND operations for a sufficiently large number of times, the graph will reach a state where each pair of $\text{rank}(u)$ and $\text{rank}(\pi(u))$ are in different levels (for u being an internal node).

Time Complexity for each FIND(u) Whenever a single FIND(u) is called, FIND is called for all nodes on the path connecting u to the current root of the tree containing u. A node v on the path do not need to pay \$1 only if $\text{rank}(\pi(v))$ and $\text{rank}(v)$ are in different levels, or $\pi(v)$ is the root. There can be at most $\log^* n$ nodes of the former type since Proposition 6 implies that the ranks of the nodes along this path is strictly increasing. There is 1 node for the latter type, which are the node whose parent is the root. Therefore, the time complexity for each FIND(u) is asymptotically given by $1 + \log^* n$ plus the amount of money paid by those nodes involved in FIND(u).

Putting together In Kruskal's algorithm, Find operation is executed for $O(m)$ times. We have seen that the time complexity for each Find(u) is asymptotically given by $1 + \log^* n$ plus the amount of money paid by those nodes involved in Find(u). The overall time complexity for those $O(m)$ Find operations is therefore $O(m \log^* n + [\text{total amount of money paid by all nodes}])$. Since we have proved that the award $O(n \log^* n)$ is sufficient, the overall time complexity is $O(m \log^* n + n \log^* n) = O(m \log^* n)$.

参考文献

John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973. 8

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975. 8