The Union-Find Data Structure

CMSC 27230: Honors Theory of Algorithms

January 18, 2023

Corresponding section of Kleinberg-Tardos: 4.6

1 Union-Find Data Structure

Consider the following situation which occurs when implementing Kruskal's algorithm. We have an undirected graph G where edges are being added one by one and we want to keep track of the connected components of G. In particular, we want to know if two vertices are in the same component or not. This can be done with the Union-Find data structure, which has the following operations.

1. $Find(i)$: Return the representative r of the connected component containing i.
2. $Union(i, j)$: Merge the connected components containing vertices i and j and choose a new representative for this connected component.

The Union-Find data structure can be implemented as follows. For each connected component, we store the vertices of each connected component as a tree where the root of the tree is the representative of the connected component and each other vertex has a pointer to its parent. The Find and Union operations can now be implemented as follows.

1. $Find(i)$: Starting from i, follow the pointer from each vertex to its parent until the root of the tree is reached.
2. $Union(i, j)$: After finding the roots of the trees containing i and j, merge the two trees into one tree by adding a pointer from one root to the other.

If we choose how to merge the components poorly, this data structure can take $\Omega(n)$ steps per operation. For example, let's say that whenever we merge two components, we choose whichever root has the smaller index to be the new root. If so, then if we the run the sequence of operations $Union(n-1, n), Union(n-2, n), \ldots, Union(1, n)$ then the resulting tree will consist of the pointers $n \to n-1, n-1 \to n-2, n-2 \to n-3, \ldots, 2 \to 1$, so it will essentially be a linked list. The operation $Union(n-k, n)$ takes $\Omega(k)$ time so the total running time is $\Omega(n^2)$.

Fortunately, the following two ideas can be used to avoid this worst-case behavior and improve the performance of the Union-Find data structure.

1. Merging by weight: When merging two connected components, we should take the root of the merged tree to be the root of whichever tree was larger. We can do this by having the root of each tree store the total number of vertices in the tree. When we merge two trees, we just add these numbers together.

2. Path compression: Whenever we run the $Find(i)$ operation and obtain the root r of its tree, we can change the pointers of i and all of the vertices which we encountered along the way to point directly to r.

It turns out that each of these ideas individually improves the performance of the Union-Find data structure to $O(logn)$ amortized time per operation. If both ideas are used, the performance improves to $O(\alpha(n))$ amortized time per operation where $\alpha(n)$ is the inverse Ackermann function. While $\alpha(n)$ is not quite constant, it is extremely close to being constant. For all practical purposes, $\alpha(n) \leq 4$.

1.1 Merging by Weight Analysis

The reason that merging by weight is effective is that it keeps the depths of the trees small. In particular, we have the following lemma.

Lemma 1.1. *If a tree rooted at a vertex i has depth d (where i itself is at depth 0) then it must have at least 2^d vertices.*

Proof. We can prove this by induction. For the base case, if the tree rooted at i has depth 0 (i.e. it just contains the vertex i) then it has size $2^d = 1$. For the inductive step, assume that all trees of depth d have at least 2^d vertices and consider a tree rooted at i which has depth $d + 1$.

Consider the point where the tree rooted at i first became depth $d + 1$. At this point, the tree T containing i must have been merged with another tree T' of depth d. Moreover, since i was chosen as the root of the merged tree, T must have at least as many vertices as T'. By the inductive hypothesis, T' has at least 2^d vertices so the merged tree must have at least $2 * 2^d = 2^{d+1}$ vertices. Later steps can only increase the size of this tree, so this completes the proof. □

By Lemma 1.1, if we use merging by weight then all trees have depth $O(logn)$ so the cost of each operation is $O(logn)$.

1.2 Path Compression Analysis

The analysis for path compression is more subtle. To get a sense for what is going on, it is a good exercise to repeatedly take a tree T, apply a union operation $Union(i, j)$ where i is a leaf of T and j is a single vertex, and take j to be the root of the merged tree. Note that this badly violates merging by weight, so applying both path compression and merging by weight will avoid this case (which is already not so bad).

We will analyze path compression using a potential function.

Definition 1.2. *Given a vertex v, define D_v to be the set of vertices (including v) which are descendants of v in the tree containing v.*

Definition 1.3. *Given a vertex v, define the rank $r(v)$ of v to be $r(v) = \lfloor log_2(D_v) \rfloor$.*

With these definitions, we can now define our potential function.

Definition 1.4. *Define the potential function ϕ to be $\phi = \sum_{v \in V(G)} r(v)$.*

The key observation is the following lemma, which says that

Lemma 1.5. *For any $Find(i)$ or $Union(i, j)$ operation, the sum of the number of steps needed and the change in the potential function is $O(log_2(n))$.*

Proof. Consider what happens when we do a $Find(i)$ operation. Observe that when we follow a pointer from a vertex u to its parent v which is not the root then after the operation, both u and v will point to the new root so D_v will be $D_v \setminus D_u$. Looking at the ranks of u and v, we have two cases.

1. If we originally had that $r(v) = r(u)$ then $r(v)$ must decrease. In this case, the cost of following the pointer from u to v is cancelled out by the decrease in $r(v)$.

2. If we originally had that $r(v) > r(u)$ then $r(v)$ may stay the same. However, since the maximum rank of a vertex is $log_2(n)$, this can happen at most $log_2(n)$ times.

We can view a $Union(i, j)$ operation as the $Find(i)$ and $Find(j)$ operations followed by merging the two trees into one tree. When we merge the two trees, the rank of the root of the merged tree can increase which increases our potential function ϕ. That said, since the maximum rank of a vertex is $log_2(n)$, this increase is at most $log_2(n)$. □

Since the potential function is always non-negative and starts at 0, Lemma 1.5 implies that the amortized cost of each operation is $O(logn)$.

1.3 Combining Approaches

As discussed above, when the merging by weight and path compression ideas are combined, this improves the performance to $O(\alpha(n))$ amortized cost per operation. The analysis for this is highly technical and we do not cover it here. For more details, see the following materials:

1. https://www.cs.princeton.edu/ wayne/kleinberg-tardos/pdf/UnionFind.pdf (these lecture notes contain a proof sketch for an $O(log^*(n))$ upper bound)

2. http://www.cs.cornell.edu/courses/cs6110/2014sp/Handouts/UnionFind.pdf

3. The original paper "A class of algorithms which require nonlinear time to maintain disjoint sets" by Robert Tarjan.