CMPSCI 611	**Advanced Algorithms**	**Fall 2001**

Lecture 8: October 1, 2001

Lecturer: Micah Adler *Scribe: Kazu Hirata and Yun Zhou*

8.1 Lecture Overview

- Review of Kruskal's algorithm
- Improving the performance of Kruskal's algorithm with a Union-Find data structure
- Simple implementation of a Union-Find Data Structure
- Better implementation of a Union-Find Data Structure
- Knapsack problem as an example of dynamic programming

8.2 Review of Kruskal's algorithm

Kruskal's algorithm is an algorithm we looked at before for finding the Minimum Weight Spanning Tree (MST) for an undirected graph G. The algorithm proceeds as follows:

8.2.1 Kruskal's Algorithm in Pseudo-Code

Given an undirected graph G with a set of vertices V and a set of weighted edges E. Let F be a forest which will contain the minimum spanning tree (MST).

```
sort the edges in E by increasing weight
F = ∅
for each edge e = (u, v) in E
    if F + e is acyclic
        F = F + e
return F
```

8.2.2 Running time

- The initial step of Kruskal's algorithm, sorting the edges takes $O(|E| \log |E|)$.
- There are $|E|$ tests for an acyclic graph which each take $O(1)$ time.
- There are $|V|$ additions of an edge, each of which takes $O(|V|)$ time.

So the total running time of Kruskal's algorithm is $O(|E| \log |E|$ (sorting) $+|V|^2$ (adding edges)). Note that if $|E| << |V|$, then the second term dominates.

8.2.3 We can do better than this

While the time to sort the edges cannot be better than $|E| \log |E|$, we can still improve the testing and adding of edges using carefully constructed data structure for storing edges.

When we introduced Kruskal's algorithm we used an array to represent the forest F and test for its acyclic nature. This choice of data structure effects the running time of the algorithm, the term $|V|^2$ in particular. We'll improve this using various versions of Union-Find.

It turns out that the data structure used to store edges has significant impact on the term $|V|^2$ of the running time. As such, we will examine various implementations of Union-Find, hoping that Kruskal's algorithm is optimized as a result.

8.3 Union-Find Data Structure

The Union-Find data structure contains a number of disjoint sets. Each of these sets contains an element designated as the "label" of the set. The Union-Find data structure also provides three functions to operate on the sets which it contains. One example of the Union-Find data structure would be

$$\begin{array}{ll} \{a, b, c\} & \text{with label } a \\ \{d, e, f\} & \text{with label } e \end{array}$$

Make-Set(v) The Make-Set function creates a set containing just v, sets the label of the set to v, and adds the resulting set $\{v\}$ to the sets stored by the Union-Find data structure. Using the example above, Make-Set(v) results in

$$\begin{array}{ll} \{a, b, c\} & \text{with label } a \\ \{d, e, f\} & \text{with label } f \\ \{v\} & \text{with label } v \end{array}$$

Union(u, v) The Union function takes the two disjoint sets, each containing u and v, respectively, replaces them with the union of the two sets, and sets the label of this new set to an arbitrary value in the set. Using the example above, Union(f, v) results in

$$\begin{array}{ll} \{a, b, c\} & \text{with label } a \\ \{d, e, f, v\} & \text{with label } e \end{array}$$

Find(v) The Find function returns the label of the set containing v. Using the example above, Find$(v) = e$ and Find$(a) = a$.

8.4 Kruskal's algorithm using a Union-Find data Structure

Our original impetus for designing the Union-Find data structure was to improve the running time performance of Kruskal's algorithm. Kruskal's algorithm using a Union-Find data structure is shown below.

8.4.1 Pseudo code

Note that if vertices u and v of an edge e are in different connected components of a forest F, then we can add edge e to the forest F without creating a cycle.

Using a Union-Find data structure to represent the connected components of the forest F, it is safe to add the edge e if Find(u) $\neq$ Find(v). The pseudo-code for Kruskal's algorithm with the Union-Find data structure is:

```
sort the edges in E by increasing weight
create a Union-Find data structure
for each vertex v in V
    Make-Set(v)
F = ∅
for each edge e = (u, v) in E
    if Find(u) ≠ Find(v)
        Union(u, v)
        F = F + e
    endif
return F
```

8.4.2 Running time

The running time of Kruskal's algorithm varies depending on what specific implementation of Union-Find data structure is used. In the following sections, several data structures and their running times will be discussed.

8.5 Simple Implementation

8.5.1 Internal Representation

In this implementation each set is represented as a linked list of nodes. Each node n contains three data elements, a name for the node, a node pointer pointing to the label of the set and a node pointer pointing to the next element of the linked list. At the end of the list the next pointer in the node is set to Null.

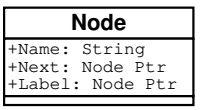
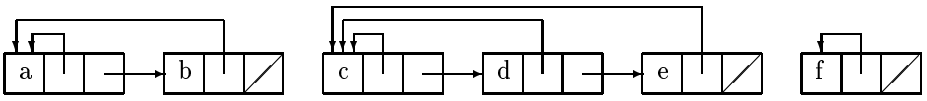


Figure 8.1: Node Class Diagram

$\{a, b\}$ with label a, $\{c, d, e\}$ with label c, $\{f\}$ with label f



8.5.2 Running time of this implementation

8.5.2.1 Running time of each operation

Make-Set(v) $\Theta(1)$ because it simply involves the creation of a single node.

Find(v) $\Theta(1)$ because it dereferences the pointer to the head of a linked list.

Union(v) O(length of two lists). When we perform a Union operation, we are appending one list to the other. Thus, we need to traverse one list to find its end and traverse the other list to relabel each node so that it will point to the node with the new common label.

8.5.2.2 Total running time

Consider n Make-Set operations followed by m Unions or Find operations. The worst case running time for this is $O(n^2 + m)$. To understand this running time, consider that there are n disjoint sets of length 1 to start with. If the longer list is always appended to the shorter list, each node must have its back pointer updated $(n-1)$ times. Since there are n nodes, this results in a $\frac{n(n-1)}{2}$ updates. The term m in the running time comes from the extra Find operations performed.

8.5.3 Improving the appending strategy

We can dramatically improve this performance by always appending the shorter list to the longer one. To do so, each node must contain a pointer to its tail node of the list the node belongs to. Also, the list must maintain its length so that one can choose the shorter list in constant time. Adding and maintaining this information does not affect the asymptotic running time of the operation.

Again consider that we have n sets to create and we are going to perform m Union or Find operations. If the shorter set is appended to the longer set, this results in at least doubling the size of the shorter set. Since we have n total elements, the size of the set resulting from a Union must be $\leq n$. Thus each node's back pointer (which is only updated when it is appended to a longer set) can only be updated a maximum of $\log n$ times. Since there are n elements, the worst case running time is now $O(n \log n + m)$.

8.5.4 Running time of Kruskal's algorithm

As before, all the edges in E are sorted by weight. This is unchanged from our previous analysis and takes $O(|E| \log |E|)$ time.

There are $|V|$ Make-Sets, each of which takes $\Theta(1)$.

There are $|E|$ Finds, each of which take $\Theta(1)$.

There are $|V-1|$ Unions, which (from our earlier analysis) take a total of $O(|V| \log |V|)$.

Thus, the total running time of Kruskal's algorithm with a simple implementation of a Union-Find data structure is $O(|E| \log |V|)$.

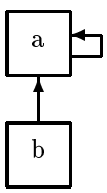
8.5.5 Faster algorithm for Minimum Spanning Trees

Although the Union-Find data structure allows us to solve the MST problem in $O(|E| \log |E|)$ it is not the fastest approach, Prim's algorithm [CLR, p. 505] utilizing Fibonacci heaps [CLR, p. 509], [CLR, p. 420], can find the Minimum Spanning Tree of a graph in $O(|E| + |V| \log |V|)$. This is a very specific result for a single problem and the Union-Find data structure is useful for a large variety of other problems.

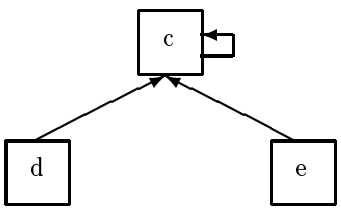
8.6 Better data structure for Union-Find

A better data structure for representing sets in the Union-Find problem is a rooted tree. This is a graph where all the nodes point towards the root. At each node in the graph is one element of the set and a pointer to the parent. The element at the root of the tree is the label of the set, and its parent pointer is self-referencing.

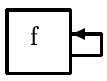
- $\{a, b\}$ with label a:



- $\{c, d, e\}$ with label c:



- $\{f\}$ with label f:

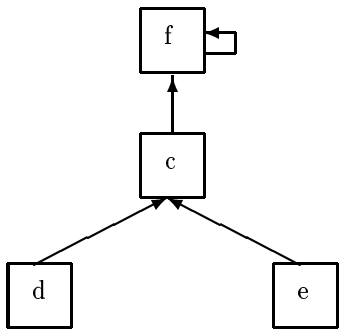


8.6.1 Description of each function

Make-Set This just involves the creation of a single node, taking $\Theta(1)$.

Find We have to chain up the tree to the root to find the label, taking Θ(depth of node d in the tree).

Union To perform a Union of the elements u and v, we first perform a Find operation on both elements. One of the root nodes of the two trees then has its parent pointer updated to point to the other tree's root. This runs in O(time for find operations + $\theta(1)$) time. For example, using the sets above, the union of f and e will produce the following rooted tree:



In the worst case, we could end up with a rooted tree of depth n after $(n-1)$ union operations. As a result, the worst case running time is $O(n \cdot m)$. This can be seen if we consider m Finds on the lowest node in the tree.

8.6.2 Improvements

The following improvements can be made to the way we use the rooted tree structure:

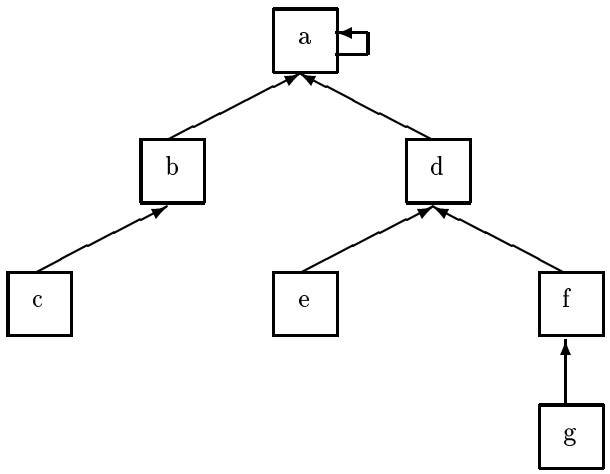
8.6.2.1 Union by size

Rather than randomly choosing the new root, always add the smaller tree to the larger tree. The worst case running time of n Make-Set operations and m Union or Find operations is now $\Theta(n \log(n))$. The proof is left as an exercise.

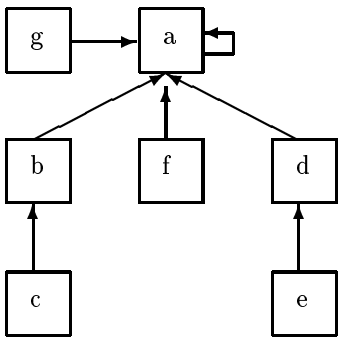
8.6.2.2 Path compression

Every time we perform a Find operation, we flatten the tree by adjusting all pointers on the path to the root so that they will directly point to the root. Since we are already chaining up the path to the root during the Find operation, changing these pointers only slows the Find operation by a constant factor.

For example, take the following rooted tree:



If we perform a Find operation on the element g, then the rooted tree will be changed to the following. Note that g and f now directly point to a.



8.6.2.3 Running time with improvements

The running time when we use the union-by-size and path-compression improvements to the Union-Find data structure is $O((n+m)\alpha(n))$, where $\alpha(n)$ is the inverse of Ackermann's function. Ackermann's function grows *very* quickly, and so the inverse grows *very* slowly. For example, $\alpha(n) \leq 4$ for $n \leq$ **BIG**.

To gain an idea of just how large the number **BIG** is, we use "towers of twos". For example:

$$\begin{aligned} 2^2 &= 4 \\ 2^{2^2} &= 16 \\ 2^{2^{2^2}} &= 2^{16} = 64K \end{aligned}$$

$$
\begin{aligned}
2^{2^{2^{2^{2}}}} &= 2^{64K} = 19729 \text{ digits in base } 10 \\
\overbrace{2^{2^{2^{2^{\cdots^{2}}}}}}^{2048\ times} &= \textbf{BIG}
\end{aligned}
$$

So for $\alpha(n)$ to be greater than 4, n must be greater than a tower of twos with height 2048.

8.6.3 Ackermann's Function

We define a sequence: $A_0, A_1, A_2...$, such that:

$$
\begin{aligned}
A_0(x) &= 1 + x \\
A_k(x) &= \overbrace{A_{k-1}(A_{k-1}(A_{k-1}(A_{k-1}(...A_{k-1})))}^{applied\ x\ times}
\end{aligned}
$$

The first few terms in the sequence of functions can be calculated easily:

$$
\begin{aligned}
A_0(x) &= 1 + x \\
A_1(x) &= \overbrace{(1 + (1 + (1 + (1 + ...(1}^{adding\ 1\ x\ times} + x))))) = 2x \\
A_2(x) &= (2(2(2(2...(2x))))) = 2^x x >> 2^x \\
A_3(x) &<< \overbrace{(2^{(2^{(2^{(...2^x)})})})}^{Tower\ of\ twos\ x\ high}
\end{aligned}
$$

As an example we take the value $x = 2$ and calculate the first 5 numbers in the sequence: $A_0(2), A_1(n)...$

$$
\begin{aligned}
A_0(2) &= 1 + 2 = 3 \\
A_1(2) &= (A_0(A_0(2)) = A_0(3) = 4 \\
A_2(2) &= A_1(A_1(2)) = A_1(4) = 8 \\
A_3(2) &= A_2(A_2(2)) = A_2(8) = 2^8 \times 8 = 2048 \\
A_4(2) &= A_3(A_3(2)) = A_3(8) << \overbrace{(2^{(2^{(...2^{2048})})})}^{Tower\ of\ 2's\ 2048\ high}
\end{aligned}
$$

Ackermann's function = Ackermann(k) = $A_k(2)$. The inverse of this is $\alpha(n)$, which is the smallest k such that Ackermann(k) $\geq n$.

8.6.3.1 Proof of $O((n + m\alpha(n))$ running time

The original proof that the running time of n Make-Set and m Union or Find operations is $O((n + m\alpha(n))$ can be found in [T75]. A better presentation of the proof can be found in [K92]. [CLR] shows a slightly weaker result of $O((n + m)\log^*(n)$.

8.7 Dynamic Programming

8.7.1 Knapsack Problem

8.7.1.1 Background

A thief with a knapack breaks into a house and steals n items, each of weight w_i and value v_i. His goal is to take items worth as much as possible while not exceeding W, a total weight that his knapsack can hold.

8.7.1.2 Formal problem description

Input: Set of n items, each of value $v_i \in Z$ and weight $w_i \in Z$ and one knapsack of capacity $W \in Z$.

Output: Subset S of the items such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is maximized.

8.7.1.3 Example 1

For the purpose of this example, we assume that $\forall i \quad v_i = w_i$.

Input: $\{6, 5, 5\}$ and $W = 10$

Output: The optimal solution is $\{5, 5\}$, but the greedy algorithm gives $\{6\}$.

Note that the greedy algorithm does not work, but the following divide and conquor algorithm works. Let a function Knap(i, j) be the weight/value of the optimal solution using only items 1 to i with capacity j. Then

Knap$(2, 5)$: $\{5\}$

Knap$(2, 8)$: $\{6\}$

Knap$(1, 5)$: $\emptyset$

References

CLR T. Cormen and C. Leiserson and R. Rivest, Matrix multiplication via arithmetic progressions, MIT Press, 1990

T75 Robert E. Tarjan, Efficiency of a good but not linear set union algorithm. Journal of the ACM, 22(2): 215-225, 1975.

K92 Dexter Kozen, The Design and Analysis of Algorithms, Springer-Verlag 1991.

BA00 Burns, and Dundas, Scribe Notes for Lecture 7, UMASS CMPSCI 611: Advanced Algorithms, Fall 2000.

RD00 Rath, and Dansong, Scribe Notes for Lecture 8, UMASS CMPSCI 611: Advanced Algorithms, Fall 2000.