CMPSCI 611 Advanced Algorithms **Fall 2000**

Lecture 7: October 3, 2000

Lecturer: Micah Adler *Scribe: Brendan Burns, Alistair Dundas*

7.1 Lecture Overview

- Review of Kruskal's algorithm
- Improving the performance of Kruskal's algorithm with a Union-Find data structure
- Simple (Naive) implementation of a Union-Find Data Structure
- Better implementation of a Union-Find Data Structure
- Analysis of the runtime of the better Union-Find Data Structure and Ackermann's function
- Discussion of a question on branchings

7.2 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:

7.2.1 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 in E
    if F + e is acyclic
        F = F + e
return F
```

7.2.2 Running time

- The initial step of Kruskal's algorithm, sorting the edges takes $|E|\log|E|$.
- 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 runtime of Kruskal's algorithm is $O(|E|\log|E|$ (sorting) $+|E|$ (testing edges) $+|V^2|$ (adding edges)).

7.2.3 We can do better than this

While the time to sort the edges (the first step of the algorithm) can't be improved since optimal sorting time is fixed at $|E| \log |E|$, we can improve our performance for the testing and adding of edges if we construct our data structure for storing edges more carefully.

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 runtime of the algorithm. It can easily be seen that we could make the addition of edges much quicker by storing the edges in the forest F as a linked list. This would make the time to add an edge to F constant. However if we are not careful the data structure that we choose, while speeding up the addition of edges will dramatically slow down our edge testing step. A data structure which hybridizes both functions is necessary. The data structure which satisfies these requirements is known as a data structure for disjoint sets, or a Union-Find data structure.

7.3 Union-Find Data Structure

The Union-Find data structure contains a number of disjoint sets. Each of these sets contains an element which has been designated as the "label" of the set. The Union-Find data structure also provides three functions to operate on the sets which it contains. For example there might exist a Union-Find data structure containing the labeled sets

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

7.3.1 MakeSet(v)

The MakeSet function creates a set of length one containing just the value specified by the parameter v, sets the label of the set to v and adds the set $\{v\}$ to the sets stored by the Union-Find data structure. Using the example above, MakeSet(v) results in

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

7.3.2 Union(u, v)

The Union function takes the two disjoint sets which contain the set elements specified by the parameters u and v and replaces them with a single set containing the union of the two sets and sets the label of this new set to an arbitrary value in the set. (e.g. for sets U, V where u is an element of U and v is an element of V, U and V are replaced by the single set $U \cup V$ and its label is some element of $U \cup V$.) Using the example above, Union(f, v) results in

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

7.3.3 Find(v)

The Find function returns the label of the set which contains the element specified by the parameter v. Using the example above, Find(v) = e and Find(a) = a.

7.4 Using a Union-Find data Structure to improve the performance of Kruskal's algorithm.

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

7.4.1 Kruskal's algorithm with a Union-Find data structure

The first thing to note is that if vertices v_1 and v_2 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 of the forest F, it is safe to add the edge e if Find(v_1) $\neq$ Find(v_2). 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 U
for each vertex v in V
    U.MakeSet(v)
F = ∅
for each edge e in E, (e = (v1, v2))
    if U.Find(v1) ≠ U.Find(v2)
        U.Union(v1, v2)
        F = F + e
    endif
return F
```

7.4.2 Runtime of Kruskal's with a Union-Find data structure

There are many different implementations of a Union-Find data structure. The runtime of Kruskal's algorithm using Union-Find is dependent on the specific implementation. In the following sections several data structures and their running times will be discussed.

7.5 Simple Implementation

7.5.1 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.

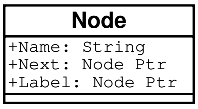


Figure 7.1: Node Class Diagram

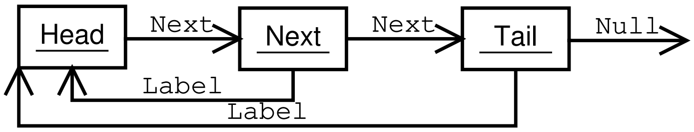


Figure 7.2: Node Example

7.5.2 Runtime of this implementation

7.5.2.1 Runtime of MakeSet

Making a set simply involves the creation of a single node which can be accomplished in constant time. So the runtime of MakeSet is $O(1)$.

7.5.2.2 Runtime of Find

Finding the label of the set containing an element is performed simply by following the pointer in the node representing the element which can be accomplished in constant time. So the runtime of Find is $O(1)$.

7.5.2.3 Runtime of Union

It might seem that a Union of two sets could be performed in constant time since two linked lists can be joined by making the tail pointer of one list point to the head pointer of the other list.

However remember that our efficient Find function is dependent upon back pointers to the label of each linked list. These pointers must be updated during the Union for the list which is appended to the tail (redefining its label). Thus the runtime is $O(|\text{list appended}|)$.

This runtime has a bad worst case time if the choice of the list to append is not performed intelligently.

Consider n MakeSet operations followed by m Unions. The worst case runtime for this is $O(n^2 + m)$. To understand this runtime consider that there are n disjoint sets of length 1 to start. If the long list is always appended to the short 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 extra $+m$ in the runtime results from the extra Find operations performed if $m > n$). This worst case running time of $O(n^2 + m)$ is worse than our previous algorithm!

7.5.2.4 A more intelligent appending strategy

The good news is that if we are slightly more intelligent in our choice of a list to append, we can dramatically improve this performance. This intelligent choice will be to always append a shorter list to a longer one.

Again consider that we have n sets and we are going to perform m Unions. If at each point the shorter set is appended to the longer set this results in (generally more than) 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 runtime is now $n \log n$.

In order to perform this optimization we have to keep track some extra information. Specifically each node must contain a pointer to its tail node and the head node of each list must maintain its length so that the choice of shorter list can be done in constant time. Adding and maintaining this information does not affect the asymptotic runtime of the operation.

7.5.3 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 $|V|$ MakeSets each of which take constant time.

There are $|E|$ Finds which of which take constant time.

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

Totaling these values, the runtime of Kruskal's algorithm with a simple implementation of a Union-Find data structure $= O(|E| \log |V|)$.

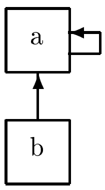
7.5.4 Faster algorithm for Minimum Spanning Trees

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.

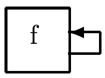
7.6 Better data structure for Union-Find

A better data structure for representing sets in the Union-Find problem is a rooted tree. This is a graph where the nodes all 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.

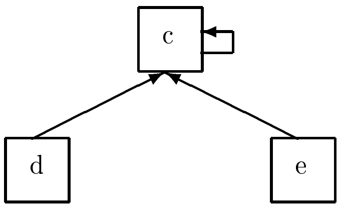
- $\{a, b\}$ with label a:



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



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



7.6.1 Running time

7.6.1.1 Runtime of Makeset

This just involves the creation of a single node. Running time is $\theta(1)$.

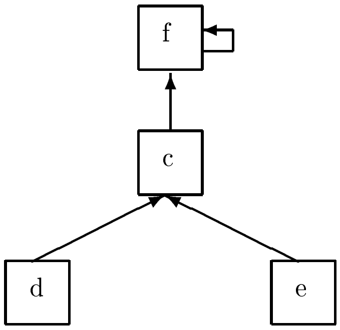
7.6.1.2 Runtime of Find

We have to chain up the tree to the root to find the label. The running time is therefore: θ(depth of node d in the tree).

7.6.1.3 Runtime of 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 has running time which is O(time for find operations + $\theta(1)$).

For example, using the sets above, the union of f and e will produce the following rooted tree:



However, if we combine sets without paying attention to which tree we choose to attach, then we could (in the worst case) end up with a rooted tree of depth n after $(n-1)$ union operations. As a result the worst case running time is $O(m \times n)$. This can be seen if we consider m Finds on the lowest node in the tree.

7.6.2 Improvements

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

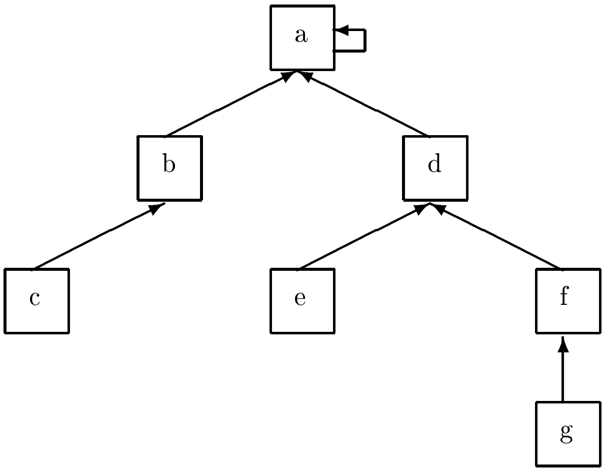
7.6.2.1 Union by size

Rather than choosing the new root without preference, always point the smaller tree to the larger tree. The worst case running time of m MakeSet operations and n Union or Find operations is now $\theta(m\log(n))$ (The proof is left as an exercise).

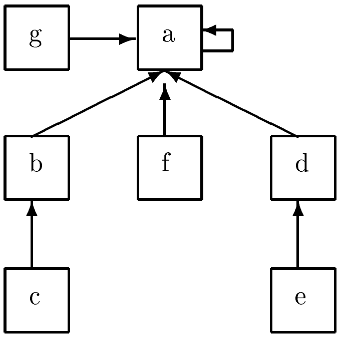
7.6.2.2 Path compression

Every time that we perform a Find operation, we flatten the tree by adjusting all pointers on the path to the root so that they 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:



7.6.2.3 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 $n \leq \mathbf{BIG}$.

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

$$\begin{aligned}
2^2 &= 4 \\
2^{2^2} &= 16 \\
2^{2^{2^2}} &= 2^{16} = 64K \\
2^{2^{2^{2^2}}} &= 2^{64K} = 19729 \text{ digits in base } 10
\end{aligned}$$

$$\overbrace{2^{2^{2^{\cdots^{2}}}}}^{2048\ times} \quad = \quad \textbf{BIG}$$

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

7.6.3 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:

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

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

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

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

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

The original proof that the running time of n MakeSet 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 : that the running time is $O((n + m) \log^*(n)$.

7.7 Branching clarification

Lecture 6 discussed branching as an application of matroid intersection, and this section clarifies that part of the lecture. The branching problem is as follows:

Input: A directed graph and a root vertex.

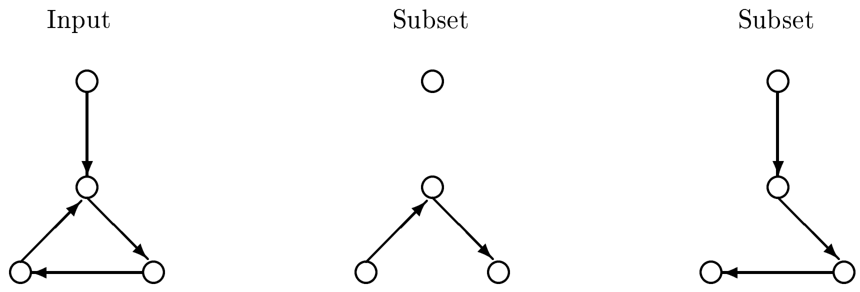
Output: A branching, if it exists, is a subset of the edges of the graph such that:

1. The corresponding undirected graph is a tree (it is acyclic).
2. All edges point away from the root.

This can be formulated as the intersection of two matroids, namely:

- A matroid with E the edges of the graph, and the subset I the sets of undirected edges forming a forest.
- A matroid with E the edges of the graph, and the subset I the sets of edges such that there is only one incoming edge to any node, and no incoming edges to the root node.

The intersection of these two matroids does not satisfy the exchange property, and therefore is not a matroid. This is demonstrated by the following diagram.



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.