

# CSE 101 Class Notes

## Disjoint Subset, Dynamic Programming

March 10, 2005

```
1  MST(G = (V, E))
2    T <- NIL
3    L <- sort(E)
4    for e = (u,v) in E
5        if not connected(T, u, v)
6            T <- T + e
7    return T
```

### Disjoint Subset Implementations

Last time we saw that Kruskal's algorithm needs a fast implementation for disjoint subsets. The simplest way to do this would be to represent the set as an array indexed by node-name, whose elements are the representative elements of a subset. To merge two subsets, we then do the following:

```
1  merge(A[1..n], x, y)
2    for i = 1 .. n
3        if A[i] = x
4            A[i] = y
```

For this approach, `find` is  $O(1)$ , but `merge` is  $O(n)$ . Since MST requires  $n - 1$  merges and  $2m$  finds, using this data structure, it will take time  $O(n^2 + m + m \log m) = O(n^2 + m \log n)$ . This is a win over the initial  $O(mn)$ , with the sorting operation dominating for dense graphs ( $m = O(n^2)$ ).

However, for sparse graphs, the  $O(n^2)$  term still dominates. To improve this, we want to make merges cheaper, perhaps at the cost of more expensive finds. A straightforward tactic would be to represent each set as a list, and merge them by adjusting pointers:

```
1  List[x] <- { linked list of just x for each x }
2  Leader[x] <- { true for each node x }
3  merge(x, y)
4    merge_lists(List[x], List[y])
5    Leader[find(x)] <- false
```

```

6
7   find(x)
8     y <- List[x]
9     while not Leader[head(y)]
10      y <- tail(y)
11     return head(y)

```

But since `find` is now  $O(n)$ , we are back to  $O(nm)$  for MST.

We can improve this by relabeling the smaller list in `merge`, and then avoiding the search in `find`. While this makes `merge`  $O(n)$  in the worst case, we claim the total cost of  $k$  merges will be less than  $O(kn)$ . To show this, we look at the sum of the sizes of all merged partitions. This equals the number of times each node is relabeled, summed over all nodes (this approach is called *amortized analysis* using the *accounting method*). If a node  $u$  is relabeled in a merge then, since it was a member of the smaller partition, after the merge  $u$ 's set has grown by at least a factor of 2. Since no set can be larger than  $n$  nodes,  $u$  cannot be merged more than  $\log n$  times, so the total cost can be no more than  $O(k \log n)$ . This is not the fastest implementation, but it is fast enough for MST, since the  $O(m \log n)$  sort now dominates for both dense and sparse graphs.

## Dynamic Programming

Fibonacci numbers are defined recursively as

$$\begin{aligned}
 F(0) &= F(1) = 1 \\
 F(n) &= F(n-1) + F(n-2)
 \end{aligned}$$

You can turn this into a recursive implementation which runs in time  $T(n) = T(n-1) + T(n-2) = F(n)$  (which is exponential). Notice that this algorithm recomputes a lot of intermediate results (e.g.  $F(n) = F(n-1) + F(n-2) = F(n-2) + F(n-3) + F(n-2)$ ). This suggests that we store the intermediate values  $F(i)$  and re-use them. Following this intuition, we can define an efficient recursive (“memoized”) implementation:

```

1   Fib <- { array 1..n of results }
2   fib(n)
3     if n < 2
4       return 1
5     else
6       fib(n - 1)
7       Fib[n] <- Fib[n-1] + Fib[n-2]
8       return Fib[n]

```

Dynamic programming combines storing intermediate results with a second idea: instead of computing the subproblems top-down, we compute them bottom-up:

```
1  fib(n)
2    Fib <- { array 1..n of results }
3    for i = 2 .. n
4      Fib[i] <- Fib[i-1] + Fib[i-2]
5    return Fib[n]
```