Eksamensnoter - Disjoint Sets and Plane Sweep

André Oskar Andersen (wpr684)

March 30, 2020

21 Data Structures for Disjoint Sets

- Some applications involve grouping n distinct elements into acollection of disjoint sets. These applications often need to perform two operations in particular: finding the unique sets that contains a given element and uniting two sets.

21.1 Disjoint-set operations

- A *disjoint-set data structure* maintains a collection $\mathcal{S} = \{S_1, S_2, ..., S_k\}$ of disjoint dynamic sets. We identify each set by a *representative*, which is some member of thes et. If we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same anaswer both times.

- We represent each element of a set by an object. Letting x denote an object, we wish to support the following operations:

  - `MAKE-SET(`x`)`: creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set
  - `UNION(`x`, `y`)`: unites the dynamic sets that contain x and y, into a new set that is the union of these two sets.
  - `FIND-SET(`x`)`: returns a pointer to the representative of the (unique) set containg x

- Since the sets are disjoint, each `UNION` operation reduces the number of sets by one. After $n-1$ `UNION` operations, therefore, only one set remains. The number of `UNION` operations is thus at most $n-1$.

An application of disjoint-set data structures

- One of the many applications of disjoint-set data structures arises in determinening the conencted components of an undirected graph.

- The procedure `CONNECTED-COMPONENTS` that follows uses the disjoint-set operations to compute the connected components of a graph. Once `CONNECTED-COMPONENTS` has preprocessed the graph, the procedure `SAME-COMPONENTS` answers queries about whether two vertices are i nthe same connected component.

21.2 Linked-list representation of disjoint sets

- A simple way to implement a disjoint-set data structure: each set is represented by its own linked list. The object for each set has attributes *head*, pointing to the first object i nthe list, and *tail*, pointing to the last object. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Within each linked list, the objects may appear in any order. The representative is the set member in the first object in the list.

- Both `MAKE-SET` and `FIND-SET` requires $O(1)$ time.
- To carry out `MAKE-SET(`x`)`, we create a new linked list whose only object is x
- For `FIND-SET(`x`)`, we just follow the pointer from x back to its set object and the nreturn the member in the object that *head* points to.

A simple implementation of union

- The simplest implementation of the `UNION` operation using the linked-list set representation takes significantly more time than `MAKE-SET` or `FIND-SET`. We perform `UNION(`x`, `y`)`, by appending y's list onto the end of x's list. The reåresentative of x's list becomes the representative of the resulting set.

21.3 Disjoint-set forests

- In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set.
- In a *disjoint-set forest* each member points only to its parent. The root of each tree contains the representative and is its own parent.
- A `MAKE-SET` operations simply creates a tree with just one node. We perform a `FIND-SET` operation by following parent pointers until we find the root of the tree. A `UNION` operation causes the root of one tree to point to the root of the other