CPS222 Lecture: Sets

last revised April 16, 2015

Objectives:

1. To introduce representations for sets that can be used for various problems
   a. Array or list of members
   b. Map-based representation
   c. Bit vectors
   d. Union-Find structure
2. To introduce basic algorithms for working with sets
   a. Merge-based algorithms for intersection, disjunction, difference
   b. Union-Find

Materials:

1. Projectable of random maze creation example
2. Handout of union/find code from program that does this

I. Introduction

A. One issue that arises in many kinds of problems is the need to represent and work with a type of information that can be understood as a mathematical set - i.e. each potential element is present exactly once or not at all, and order is not important.

   1. The fundamental operation on a set is the contains test - does the set contain a specific element?

   2. But other operations such as union, intersection, difference etc. are often important.

B. There are a variety of different ways of representing sets, with the choice depending on the nature of the information and how it is to be manipulated.

1. A simple representation is to use a container such as an array or list that holds the set elements, which can, in this case, be of any type.

   a) Example: the set { Aardvark, Buffalo, Cat, Dog, Elephant } might be represented by a sequence such as an array or list whose elements are the strings "Aardvark", "Buffalo" etc.

   b) Of course, this representation implicitly creates an order for the elements, but the order is generally regarded as unimportant

   c) Of more concern is the fact that the contains() operation will involved iterating over all the elements in the set - which is O(set size)

2. Both the Java library and the C++ STL incorporate set containers, which are actually implemented as maps, with each key simply being mapped to itself.

   a) As you recall, Java offers two implementations: `HashSet` and `TreeSet`, which are actually realized by `HashMap` and `TreeMap`, respectively. Of course, the latter does, in fact, keep elements in an order and thus requires that the element type support comparison (e.g. that it is something like an `Integer` or a `String`.)

   b) C++ draw the same distinction, but the uses the name `unordered_set` for the version that uses hashing and reserves the name `set` for the version that maintains order (and hence must have an element type for which comparison is defined, either builtin by < or user-defined.

   c) Either representation makes a quick implementation of contains() easy (its just key lookup in the map, hence O(1) or O(log set size).

3. For representing sets whose elements are drawn from a fixed (and usually small) universe, a representation known as a bit vector can be used.

a) Example: suppose we are interested in sets of letters of the alphabet - e.g. vowels = { A, E, I, O, U }, labial consonants (consonants made with the lips) = { B, F, M, P, V } etc. In this case, our universe has 26 members.

b) To represent sets as a bit vector, we use a vector of bits equal in length to the size of the universe (e.g. a vector of 26 bits for letters of the alphabet), with one position explicitly corresponding to each element of the universe (e.g. the leftmost bit might represent A, the one next to it B ... the rightmost bit Z).

   The representation for a set has a 1 in a given position if the corresponding element is in the set, and 0 if it is not - e.g. the set vowels might be represented by the bit vector `10001000100000100000100000`.

c) This supports a fast implementation of contains - simply look to see if the bit corresponding to the desired value is set.

d) But, of course, this is limited to cases where the set of possible values is small and can easily be mapped to bit positions.

4. We will look at one more structure - the union-find structure - later.

II. Standard Set Operations

A. Many operations on sets make use of the standard set operations union, intersection, difference and - more rarely - negation (which is only meaningful if the set is based on a finite universe.)

   Example: consider the search operations offered in many contexts. Operations such as and, or, and not map to set operations - e.g. the set of documents containing two different keywords is the intersection of the two sets of documents containing each keyword.

B. If the sets are represented as sequences or maps, one (computationally expensive) way to perform these operations would be to make use of the contains operation.

1. Example: `A ∩ B` could be implemented as follows: iterate through the elements of set A. For each element, test if B contains it, and include it in the result only if it does.
2. This approach could be as bad as $O(n^2)$ if the contains operation is implemented by iterating through all the elements of a set to see if it is there. If a tree-based representation for sets is used, it would be O(n log n)), and if a hashtable is used for sets, it might be O(n), though possibly with a large constant of proportionality.
3. If some sort of ordered representation is used (a tree or if an array or list is kept in some order), a much faster implementation is possible based on merging.

   a) What we do is iterate through both sets in order. Let $\texttt{current}_A$ be the current element from set A, and $\texttt{current}_B$ the current element from set B. If we have run out of elements in a set, let current for that set be a "manufactured" element that tests greater than any real element. Then

   while not past the end of both sets:
   if $\texttt{current}_A == \texttt{current}_B$:
   include the element in the result if the operation is union or intersection, but not if it is difference
   advance to the next element in both sets
   else if $\texttt{current}_A < \texttt{current}_B$
   include $\texttt{current}_A$ in the result if the operation is union or difference, but not if it is intersection
   advance to the next element in set A only
   else (it must be that $\texttt{current}_A > \texttt{current}_B$)
   include $\texttt{current}_B$ in the result if the operation is union, but not if it is intersection or difference
   advance to the next element in set B only

b) Example: calculate the intersection of { Aardvark, Buffalo, Zebra } and { Buffalo, Cat }

$current_A$ is Aardvark
$current_B$ is Buffalo
Since $current_A < current_B$, advance to the next element in A,
(so $current_A$ is now Buffalo)
Since $current_A == current_B$, include Buffalo in the result and advance to the next element in both A and B.
(so now $current_A$ is Zebra and is $current_B$ Cat)
Since $current_A > current_B$, advance to the next element in B,
(so $current_B$ is now ∞)
Since $current_A < current_B$, advance to the next element in A,
(so $current_A$ is now ∞)
Stop - result is { Buffalo }

c) Of course, this approach is O(n)

4. If the set is represented by a bit vector, basic set operations can be done even more efficiently by using bitwise logical operations - e.g. set union is bitwise or; set intersection is bitwise and, set negation (possible with a finite universe) is bitwise complement, and difference is bitwise and with the complement.

   Example: Represent the consonants and vowels by bit vectors; { 01110111011111011111011111 and 10001000100001000000100010 } show how to find letter than is both by set intersection.

III.Union-Find

A. One category of operations that shows up in a number of places is based on the notion of a partition of a set. A partition is a set of subsets of the original set such that each element of the original set shows up in exactly one subset.

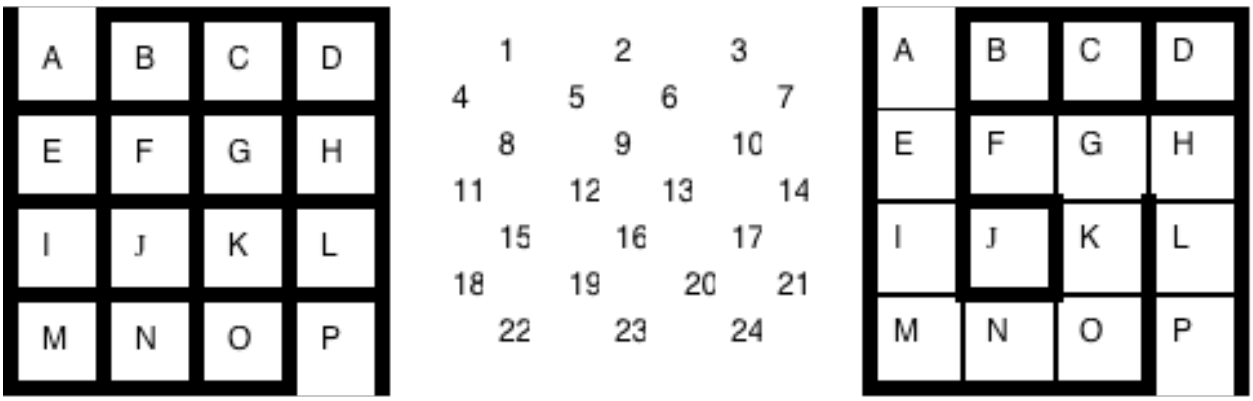
1. Example: create a partition of the class based on academic year

2. Example: because of double majors, it would not be possible to create a partition of a class based on academic major (since some people would show up in two subsets) - though it would be possible if we used first major as the basis for partitioning.

B. For a partition, two operations are especially useful:

1. Find the partition a particular element belongs to.
2. Take the union of two subsets so that all the elements in either subset now belong to united subset.

C. An example where this is useful: suppose we wanted to randomly generate a solvable maze. One way to do this would be to start with a maze structure in which all the possible walls were present, and then begin to remove walls at random until there was a path from start to finish.



1. Example. Starting with the simple maze on the left - with start cell A , we can create the maze on the right by generating random number, using the assignment of numbers to walls shown in the middle, and removing walls until there is a path from start to finish. (The example required removal of walls numbered 4, 10, 21, 22, 18, 9, 23, 11, 20, 13, 14 in that order.)
   PROJECT

2. Of course, if one were to do this, one would need some way of testing, at each step, whether a path exists.

a) A good way to do this would be to form a partition, with each cell initially in its own set. Each time a wall is removed:

(1) Find the subset for each of the two cells that were separated by the wall.

(2) If the subsets differ, replace them by their union.

(3) If the two cells were already in the same subset, and we don't want to have cycles in the maze, we can not remove the wall.

b) The result is that, at any time, each cell is in a set consisting of the cells that are reachable from it.

c) When both the start and the finish cells are in the same set, we can stop.

3. Can you think of another place where we have seen something similar?

ASK

In Kruskal's shortest path algorithm, we need to test when we're adding an edge to see whether it would create a cycle, which occurs if it joins two vertices that are part of the same connected component.

D. With the set representations we have considered thus far, the find operation is computationally expensive because we need to test each of the sets to see if the element in question is in it.

1. A much faster solution arises if we represent sets by trees in which the pointers go the other way - i.e. each element node points to its parent, rather than the other way around.

a) We could have a special node to label the set, or we could just let one of the elements serve that purpose, with a pointer to itself.

Example: The we could picture the cells after the removal of walls 4, 10, 21, 22 this way:

```
A   B   C   D   F   G   I   J   K   L   M   O
|                   |               |   |
E                   H               P   N
```

Note that find now requires just one step.

b) If the element node in question has no parent, it is the name of the set; otherwise the name of the set is the node it points to.

Example: A belongs to set A; H belongs to set G.

2. To perform union, we can simply make one set a subset of the other by making the root of the subset point to the root of the other.

a) Of course, if we do union this way we may have to iterate up a tree to find the root of a set - e.g. if we take the union of sets A and G, we would get

```
  A
 / \
E   G
    |
    H
```

H belongs to set A, but we had to iterate through G to find this out.

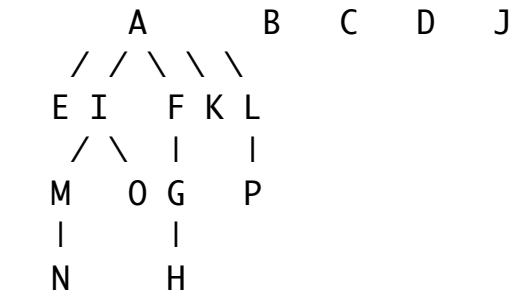
b) A heuristic called union by size says to make the set containing the smaller number of elements the subset. (This minimizes the need for iteration)

Example: our partition after removing all but the last 2 walls (13, 14), using union by size, is

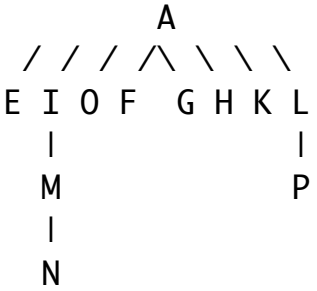
```
    A   B   C   D   J   L
 / / \ \                |
E I   F K               P
 / \  |
M   O G
|    |
N    H
```

(1) Removing wall 13 has no effect, since it the cells it separates (G and K) are already part of the same group.

(2) Removing wall 14 (which separates H and L) produces this result:



c) It is also possible to do path compression during iteration - so that all nodes we pass through are made direct children of the root. If we did this, when we were done, set A would look like this



This reduces - but does not eliminate - the need for iteration in the find operation.

3. A program that uses this approach

a) DEMO

b) HANDOUT Union/Find code