MA/CSSE 473 Day 39

Disjoint Set Implementation

Dijkstra

MA/CSSE 473 Day 39

- HW 15 Due Friday (it's a little different!)
- Fill out the Course evaluation form
  - If everybody in a section does it, everyone in that section gets 10 bonus points on the Final Exam
  - I can't see **who** has completed the evaluation, but I can see **how many (current: 5/16, 5/23)**
- Final Exam Wednesday evening, Nov 17
- **Student Questions**
- Prim data structures and detailed algorithm
- Kruskal data structures and detailed algorithm

Final Exam Format

- **Part 1:** You may use textbook, two 2-sided 8.5"x11" sheets of paper, calculator
- **Part 2:** You may also use your homework assignments, notes, and quizzes, things I have posted for the course, programs that you have written
- Receive both parts at the beginning of the exam; work on questions in any order, turn in part 1 before using additional part 2 resources

Disjoint Set Representation

- Each disjoint set is a tree, with the "marked" element as its root
- Efficient representation of the trees:
  - an array called *parent*
  - parent[i] contains the index of i's parent.
  - If i is a root, parent[i]=i

i	1	2	3	4	5	6	7
parent[i]	1	5	7	5	5	7	7

Using this representation

- makeset(i):

```
def makeset1(i):
    parent[i] = i
```

- findset(i):

```
def findset1(i):
    while i != parent[i]:
        i = parent[i]
    return i
```

- mergetrees(i,j):
  - assume that i and j are the marked elements from different sets.

```
def mergetrees1(i,j):
    parent[i] = j
```

- union(i,j):
  - assume that i and j are elements from different sets

```
def union1(i,j):
    mergetrees1(findset1(i), findset1(j))
```

Q1

Analysis

- Assume that we are going to do n makeset operations followed by m union/find operations
- time for makeset?
- worst case time for findset?
- worst case time for union?
- Worst case for all m union/find operations?
- worst case for total?
- What if m < n?
- Write the formula using the min function

Q3

Can we keep the trees from growing so fast?

- Make the shorter tree the child of the taller one
- What do we need to add to the representation?
- rewrite makeset, mergetrees.

```
def makeset2(i):
   parent[i] = i
   height[i] = 0
```

```
def mergetrees2(i,j):
   if height[i] < height[j]):
         parent[i] = j
   elif height[i] > height[j]:
      parent[j] = i
   else:
      parent[i] = j
      height[j] = height[j] + 1
```

- findset & union are unchanged.
- What can we say about the maximum height of a k-node tree?

Theorem: max height of a k-node tree T produced by these algorithms is $\lfloor \lg k \rfloor$

- Base case…
- Induction hypothesis…
- Induction step:
  - Let T be a k-node tree
  - T is the union of two trees:
    T_1 with k_1 nodes and height h_1
    T_2 with k_2 nodes and height h_2
  - What can we about the heights of these trees?
  - Case 1: $h_1 \neq h_2$. Height of T is
  - Case 2: $h_1 = h_2$. WLOG Assume $k_1 \geq k_2$. Then $k_2 \leq k/2$. Height of tree is 1 + h2 ≤ …

Q2

Worst-case running time

- Again, assume n makeset operations, followed by m union/find operations.
- If m > n
- If m < n

Speed it up a little more

- **Path compression:** Whenever we do a findset operation, change the parent pointer of each node that we pass through on the way to the root so that it now points directly to the root.
- Replace the height array by a rank array, since it now is only an upper bound for the height.
- Look at makeset, findset, mergetrees (on next slides)

Q3

Makeset

This algorithm represents the set {*i*} as a one-node tree and initializes its rank to 0.

```
def makeset3(i):
   parent[i] = i
   rank[i] = 0
```

Findset

- This algorithm returns the root of the tree to which i belongs and makes every node on the path from i to the root (except the root itself) a child of the root.

```
def findset(i):
   root = i
   while root != parent[root]:
      root = parent[root]
   j = parent[i]
   while j != root:
      parent[i] = root
      i = j
      j = parent[i]
   return root
```

Mergetrees

This algorithm receives as input the roots of two distinct trees and combines them by making the root of the tree of smaller rank a child of the other root. If the trees have the same rank, we arbitrarily make the root of the first tree a child of the other root.

```
def mergetrees(i,j) :
   if rank[i] < rank[j]:
      parent[i] = j
   elif rank[i] > rank[j]:
      parent[j] = i
   else:
      parent[i] = j
      rank[j] = rank[j] + 1
```

Analysis

- It's complicated!
- R.E. Tarjan proved (1975)*:
  - Let t = m + n
  - Worst case running time is Θ(t α(t, n)), where α is a function with an *extremely* slow growth rate.
  - Tarjan's α:
    - $\alpha(t, n) \leq 4$ for all $n \leq 10^{19728}$
- Thus the amortized time for each operation is essentially constant time.

* According to *Algorithms* by R. Johnsonbaugh and M. Schaefer, 2004, Prentice-Hall, pages 160-161

Q4

Dijkstra's Algorithm

- For a selected vertex, (call it ***start***) in a connected, weighted graph G, find a shortest (minimum total weight) path from *start* to each other vertex
- Assumption: All weights are positive

Dijkstra's algorithm approach

- The shortest path from vertex v to vertex w is either
  - empty, if v = w, or
  - obtained by adding an edge (u, w) to the end of a shortest path from v to u.

Dijkstra's algorithm start-up

- Begin with a subgraph that consists of only the starting vertex.
- Among all edges (start, v), choose the one with the smallest weight, and add it (along with the edge that connects it) to our subgraph. Clearly (start, v) is the shortest path from *start* to v.

Dijkstra's algorithm continues

- Loop:
  Among all vertices not in a path yet, and which are also adjacent to a vertex that is in the path, choose one with minimal path length from start, and add it to the path, along with the edge that connects it to a path.
- Which algorithm that we have seen does this closely resemble?

Example:

```
g = AdjancencyListGraph(
           [[1, [(2, 4), (3, 2), (5, 3)]],
            [2, [(1, 4), (4, 5)]],
            [3, [(1, 2), (5, 6), (4, 1), (6,3)]],
            [4, [(2, 5), (6,6), (3,1)]],
            [5, [(1, 3), (3,6), (6,2)]],
            [6, [(5, 2), (3, 3), (4, 6)]]
            ])
```

Q6

Dijkstra's algorithm data structures

- A parent array as in Prim's algorithm.
- A minheap that stores "unpathed" nodes as keys and minimum path lengths (as extensions of a minimum path that has already been discovered) as weights.
  - Use an indirect binary heap, just as we did for Prim.
- A cost array that stores the minimum path length to each vertex
  - Not strictly necessary; could reconstruct form adjacency list and parent array.

Dijkstra's algorithm details

```
def dijkstra(adj, start):
    """ parent[v] = parent of v in shortest path to v rooted at start """
    n = adj.length()              # vertices in graph
    key = [None] + [INFINITY]*n  # later they will be decreased
    parent = [None] + [0]* n
    cost = [None] + [0]*n
    key[start] = 0
    heap = MinHeap(key)  # non-infinity value in heap represents fringe verte
    for i in range(1, n+1):
        minCost = heap.minKey()
        v = heap.delete()
        edges = adj.getList(v)  # all vertices adjacent to v
        for edge in edges:  # an edge is a list of: other vertex and weight
            w = edge[VERTEX]
            if heap.isIn(w) and minCost + edge[WEIGHT] < heap.keyVal(w):
                parent[w] = v
                cost[w] = minCost + edge[WEIGHT]
                heap.decrease(w, cost[w])

    return parent, cost
```

Q7-8