Union-Find
Greedy algorithms, greedy heuristics

CS 2860: Algorithms and Complexity

Magnus Wahlström and Gregory Gutin

February 26, 2018

Random-order connectivity: Union Find

“Online” / “dynamic” connectivity

- Graph $G = (V, E)$ arrives online (bit-by-bit):
  - Initially, have $G = (V, \emptyset)$
  - Then edges $uv \in E$ arrive, one by one, unknown order
- Want to know, at any point in time:
  What are the current connected components?
- Want to know this (data structure),
  without having to compute it when asked

Connectivity and partitions

- For every graph $G = (V, E)$, the connected components partition V:

$$V = V_1 \cup V_2 \cup \ldots \cup V_k, \text{ all disjoint}$$

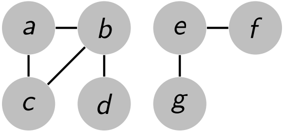
- The sets V_1, V_2, ... are called blocks of the partition
- The initial empty graph has the partition:

$$V = \{v_1\} \cup \{v_2\} \cup \ldots \cup \{v_n\} \text{ (singleton sets)}$$

- Adding an edge uv between two components causes the components to merge:

$$V_i \leftarrow V_i \cup V_j; \text{ partition } V_j \text{ deleted}$$

Dynamic connectivity: Example



Edges added	Partition						
None	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$
bd	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$
eg	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$	
ac	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$	
ab	$\{a, b, c, d\}$				$\{e, g\}$	$\{f\}$	
ef	$\{a, b, c, d\}$				$\{e, f, g\}$		
bc	$\{a, b, c, d\}$				$\{e, f, g\}$		

Operations needed

- Boolean Connected(u,v): Are *u* and *v* in the same block?
- Merge(u,v): Join the blocks containing *u* and *v* together

More commonly given as:

- Find(u): Return a representative of the set containing *u*
- Union(u,v): Merge the blocks containing *u* and *v*
- Then Connected(u,v) turns into (Find(u)==Find(v))

Our first attempts

- A partition is a collection of sets
- We know how to implement sets
  - As HashSet if we need fast membership
  - As ArrayList if we only need iteration
- Find ("which set is this vertex contained in?") sounds like a Map or just an array (if vertices have "integer names")
- But Union (merge two sets) will be slow
  - "Put set B into set A [and update the vertex map/array]" takes $\mathcal{O}(|B|)$ time
- Can get $\mathcal{O}(n \log n)$ total time, but not better

Better idea: Parent-pointer trees

Implement the structure as a forest

- Connected component = rooted tree
- "Name" of component = its root vertex

Basic operations:

- Find(u):
  1. Start at u in forest
  2. Walk upwards until you reach the root r
  3. Return name/index of r
- Union(u,v):
  1. Find roots r_u, r_v
  2. If $r_u \neq r_v$, add an arc $r_v \rightarrow r_u$

Room for improvements

- Consider the sequence:
  - Union(1,2)
  - Union(1,3)
  - Union(1,4) ...
- Claim: Total time $\Theta(n^2)$
  - Always hangs tall tree under new root
  - After Union(1,i), the tree has i levels
  - Computing Find(1) afterwards takes i steps
- Plenty of room for improvements!

Improvement 1: Merge by rank

Idea 1: Avoid deep trees by merging small trees into large trees

- Initialisation: For each vertex $i = 0, 1, \dots, n-1$:
  - parentOf[i]=i
  - rank[i]=0
- Union(u,v) step:
  1. Compute root_u=Find(u) and root_v=Find(v)
  2. Skip if root_u == root_v
  3. If rank[root_u] < rank[root_v]:
     - Set parentOf[root_u]=root_v
  4. Otherwise:
     - Set parentOf[root_v]=root_u
     - If rank[root_u] == rank[root_v], set rank[root_u] += 1
- Maximum tree depth $\mathcal{O}(\log n)$

Improvement 2: Path flattening

- Idea: When we're walking up the tree,
  we might as well reconfigure it to be shorter
- Change to Find(u) (only):
  1. If parentOf[u]==u: Return u
  2. Otherwise:
     - 2.1 Let root = Find(parentOf[u])
     - 2.2 Set parentOf[u]=root
     - 2.3 Return root
- Observe:
  - By the recursion, every node on the way gets reattached to have the root as parent (not just the leaf node)

Running time

- For an arbitrary sequence of Union and Find operations:
  1. Original (no improvements): worst-case $\Theta(n^2)$
  2. Improvement 1 (short tree into tall tree): worst-case $\Theta(n \log n)$
  3. Improvements 1+2: Worst-case $\Theta(n \cdot \alpha(n))$, where $\alpha(n)$ is a ridiculously slow-growing function called inverse Ackermann
- How slowly growing?
  - $\alpha(7) = 2$, $\alpha(61) = 3$
  - $\alpha(n) \leq 4$ for $n \leq 2^{2^{2^{65,536}}}$
  - But $\alpha(n)$ is growing, and $\Theta(n \cdot \alpha(n))$ is tight!
- Historical curiosity:
  - The bound $n\alpha(n)$ was shown in 1975–1980 (Tarjan)
  - Before that, we "only" knew the bound $\mathcal{O}(n \log^*(n))$, where $\log^*$ is defined iteratively: $\log^* x = 0$ if $x \leq 1$ and $= 1 + \log^*(\log x)$, otherwise.
  - The function $\log^*(n)$ grows much faster: $\log^*(n) = 5$ already for $n = 2^{65,536} \approx 10^{20,000}$

Greedy algorithms

Greedy algorithms

- Kruskal and Prim are examples of greedy algorithm design principle
- Construct a solution to a problem by
  1. Find the cheapest decision to make right now
  2. Stick with it; never change your mind
  3. Keep making locally optimal decisions, without contradicting your previous ones, until solution is complete
- Comes in two flavours
  1. Greedy algorithms
  2. Greedy heuristics

Algorithms and heuristics

- To us, an algorithm must have guarantees of success
  - Must always find a solution
  - If optimisation problem (e.g., with weights), must find optimal (e.g., cheapest) solution
  - Alternatively, give precise guarantees about (non-optimal) solution quality ("approximation algorithm")
- Greedy algorithm:
  - A greedy-type solution to an optimisation problem that always finds the true (global) optimum in the end
- Heuristic:
  - Rule of thumb optimisation: usually works, sometimes fails
  - E.g. greedy heuristic: "Try grabbing the cheapest item every time, maybe it works out well in the end"

Greedy: Cases

1. Problems where (some) greedy approach works perfectly
   - Simple problems ("pick the best five items out of many")
   - Min-cost spanning trees
   - Some clever greedy cases (room bookings, next slide)
   - ★Matroids
2. Problems which have some efficient solutions, but not greedy
   - Dormitory assignment (a.k.a. graph matching)
   - Some traffic scheduling (a.k.a. max flow)
   - Many more
3. Problems with only inefficient perfect solutions (e.g., NP-hard), where maybe greedy heuristics will work for you
   - Surprisingly often, it will
   - But it can also work really badly

Greedy or not? Two problems.

1. Making change
   - Input is coin denominations (e.g., 1, 2, 5, 10, 20, 50 pence) and target value (e.g., 67p)
   - Want to pay using fewest possible coins (example: 67=50+10+5+2 gives 4 coins)
   - Clarification: Have unbounded number of each coin
2. Room scheduling
   - Input: A list of booking requests for a room (e.g., "from 1:00pm to 1:45pm", "from 1:30pm to 3:30pm", . . .)
   - Want to find a largest set of non-conflicting bookings
   - Clarification: May treat a request as a pair (start, end) of integers

Result

1. Making change:
   - Not greedy (e.g., coins of 1p, 20p, 30p, target=40p)
   - But greedy works for some coin systems
2. Room scheduling:
   - Greedy – but only with the right greediness!
   - Does not work: "Select earliest-starting bookings", "select shortest bookings", ...
   - Works: "Select earliest-ending booking" (then continue)