4. Amortized analysis and merge-find data structures

4.1 What is amortized analysis (A.A.)?

Setting Suppose we are dealing with an algorithm or data structure where

- most operations are fast, i.e. "cheap"
- a few, sparsely used operations are slow, i.e. "expensive"

Key Ideas

- express every expensive operation as a finite sequence of k cheap operations
- show that the worst-case cost for the *entire* sequence of k cheap operation is lower than the *sum* of worst-case costs for each of the k *individual* operations, i.e.

$$cost(\sum_{i=1}^{k} operation_i) \leq \sum_{i=1}^{k} cost(operation_i)$$

where *operation* consists of k operations (from $operation_1$ to $operation_k$).

Definition *Amortized Analysis* is a set of techniques we can employ to estimate the worst-case cost for an *entire* sequence consisting of k operations.

Example suppose we are dealing with the following individual operations:

- PUSH (worst case time is $\Theta(1)$)
- POP (worst case time is $\Theta(1)$)
- MULTIPOP(j) (worst case time is $\Theta(n)$ for $j \leq n$ elements)

MULTIPOP pops $1 \leq j \leq n$ elements, where there are n elements in total.

Group Work What is the worst case time for a sequence of n called to a combined mix of PUSH, POP, and MULTIPOP?

Answer Every one of the n elements can be pushed/popped out at most once, so the overall worst-case time is $\Theta(n)$. For example, there cannot be n calls to MULTIPOP (at a cost of $\Theta(n)$ each) as there are only a *total* of n elements to pop, i.e. the overall worst-case time is *not* $\Theta(n^2)$.

Conclusion *Key Idea:* In order to get accurate boundaries for the execution time for an *entire* sequence of k operations, we need to take into account how those k operations influence each other.

4.2 The potential method

Key Ideas

- we are performing a sequence of k individual operations, denoted

$$op_i \text{ with } i \in \{1, 2, \ldots, k\}$$

- each operation op_i is assigned an *amortized cost* ($cost_{am}(op_i)$) which may differ from the *actual cost* ($cost_{real}(op_i)$) with the $operation_i$ is actually executed.
- *Note:*
  - $cost_{am}(op_i)$ may be $<$, $=$, $>$ than $cost_{real}(op_i)$.
  - the am. cost typically constitutes our best guess of the cost of operation $op + i$ in situations where we cannot specify the worst-case running time of operation op_i as a function of input site n up-front.
- when we execute the sequence of k operations and we are dealing with operation op_i, we:
  1. we "pay" $cost_{am}(op_i)$
  2. if $cost_{real}(op_i) < cost_{am}(op_i)$ {
     save $cost_{am}(op_i) - cost_{real}(op_i)$ in "bank" (to be spent later, if needed)
     }
     else {
     take $cost_{real}(op_i) - cost_{am}(op_i)$ from bank if our balance allows this.
     (the bank balance at any time has to be ≥ 0!!)
     }
- in the above, interpret
  - the bank balance, which is a function of i, $1 \leq i \leq j$, as overall cost of the algorithm up to operation i
  - D_i = data structure after operations 1 to i, $i \leq i \leq k$
  - $\Phi(D_i)$ = bank balance or total cost of executing operations 1 to i, $i \leq i \leq k$ (function Φ is also called a *potential function*)

Properties of the potential function

1. $\Phi(D_i) \geq 0 \ \forall i \in \{1, \ldots, k\}$
   interpretation: cannot borrow execution time or cost if we haven't correctly budgeted for it by amortized costs up front ("bank cannot go broke")
2. $\Phi(D_0) = 0$
3. $\Phi(D_i) = \Phi(D_{i-1}) + cost_{am}(op_i) - cost_{real}(op_i)$ for all $i \in \{1, \ldots, k\}$

Logical flow in the following and in practice:

- Start by defining a potential function Φ and then determine the amortized costs by enforcing the above properties of Φ rather than first defining the amortized costs which are difficult to guesstimate up front.

- if we are dealing with a sequence of n operations, we want to show that

$$\sum_{i=1}^{n} cost_{real}(op_i) \leq \sum_{i=1}^{n} cost_{am}(op_i)$$

as we then know that the worst case running time for this sequence of n operations is

$$O(\sum_{i=1}^{n} cost_{am}(op_i))$$

Example
- stack with operations PUSH, POP, MULTIPOP, n elements, $k \leq n$ ops.
- choose $\Phi(D_i) =$ size of D_i
- want (from previous list):
  1. $\Phi(D_i) \geq 0 \ \forall i \in \{1, \ldots, k\}$
  2. $\Phi(D_0) = 0$
  3. $\Phi(D_i) = \Phi(D_{i-1}) + cost_{am}(op_i) - cost_{real}(op_i)$

Group Work Choose $cost_{am}$ so 1. and 3. above are satisfied.

Idea
- For $1 \leq i \leq k$, consider $op_i =$ PUSH in 3., i.e.

$$\Phi(D_i) = \Phi(D_{i-1}) + cost_{am}(PUSH) - cost_{real}(PUSH)$$
$$cost_{am}(PUSH) = (\text{size of } D_i) - (\text{size of } D_{i-1}) + 1 = 2$$

- for $op_i =$ POP, get $cost_{am}(POP) = 0$ in a similar manner
- for $op_i =$ MULTIPOP(j) with $i \leq j \leq$ (size of D_{i-1})

$$\begin{aligned}(3) &= costam(MULTIPOP(j)) \\ &= (\text{size of } D_i) - (\text{size of } D_{i-1}) + cost_{real}(MULTIPOP(j)) \\ &= (\text{size of } D_i) - (\text{size of } D_i + j) + cost_{real}(MULTIPOP(j)) \\ &= -j + cost_{real}(MULTIPOP(j)) \\ &= n - j\end{aligned}$$

4.3 The union-find data structure

Goal We wish to create a data structure that supports the following operations:

1. return the name of the set that contains element x (find(x). If two elements x and y, $x \neq y$, are in the same set then find$(x) =$ find(y).
2. merge the data structures corresponding to two sets A and B into one data structure (union(A, B))

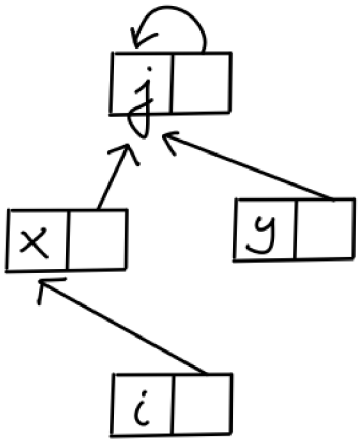
3. create a data structure where every element $x \in S$ corresponds to a separate set (makeUnionFind(S))

Motivation We would like to define a data structure which facilitates the operations inside the while-loop of Kruskal's algorithm.

Ideas Use a tree-like data structure where every node v in the tree corresponds to an element of the corresponding set, $v \in S$, and where each node v in the tree has a corresponding pointer which points to the name of the set that contains v.

Name every set after one of its elements.

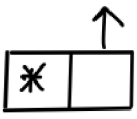
Example: $S = \{i, j, x, y\}$ is given name $j \in S$



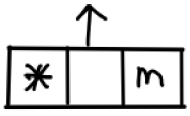
1. if we are dealing with a set of $|S| = n$ elements, the find(x) operation requires up to $O(n)$ to follow the pointers starting at x to reach the node (j in the above example) after which the set containing x is named.

   To further reduce the time for find(x), we will in the following adopt the name of the larger set as the name of the union of two sets A and B.

   Replace every

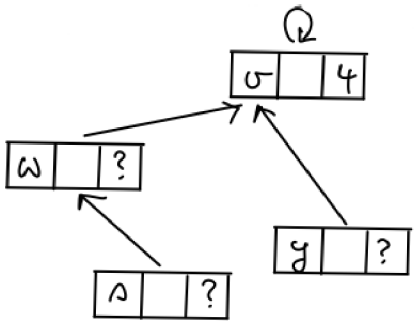


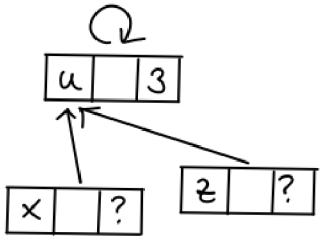
   above by



   where m denotes the size of the set to which $\star$ belongs ($m = 4$ in the above example).

2. When we want to merge two data structures, one representing set A and one representing set B, using union(A, B) we
   - check if set A or B is larger (this takes $O(1)$, see 1.)
   - if, say $|A| > |B|$, and $v \in A$ is the name of A and $u \in B$ is the name of B, we adopt the name v for $A \cup B$ and merge the two data structures by only adjusting the pointer of $u \in B$ to point to $v \in A$.

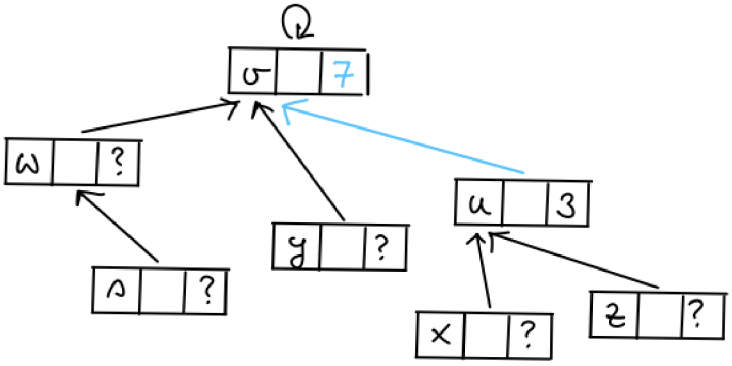




$A = \{v, w, n, y\}$
name of A is v

$B = \{u, x, z\}$
name of B is u

after union(A, B):



= new

$A \cup B = \{v, w, n, y\} \cup \{u, x, z\}$

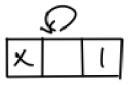
The name of $A \cup B$ is v and we only need to update the number of elements of set $A \cup B$ at node v.

? denotes entries that have been set before, but that we do not need to know or change when executing union(A, B)

union(A, B) requires $O(1)$ time.

3. creating a data structure where every element $x \in S$ corresponds to a separate set (make UnionFind(S)) takes $O(n)$ time as we need to create n separate entities

of the following kind where $|S| = n$:



Insight 1 1. Revisited: We already know that find(x) requires at most $O(n)$ time. If we keep track of the size of elements in a set (using a size field for every node) and if we adopt the convention that the union of any two sets is named after the largest set (see 2.), it follows that every time the name of the set containing node x changes, the size of the current set at least doubles in size.

Insight 2 The cost of find(x is equal to the number of pointers it takes to read the node after which the set is named, i.e. the number of times that the set containing node x changes its name. As the set containing node x starts with a size of 1 (x itself), and is never larger than n , it can double its size at most $k = \log_2(n)$ times as $2^k = n$.

$\rightarrow$ There can be at most $\log_2(n)$ name changes for the set containing x.

$\rightarrow$ find(x) requires $O(\log(n))$ time.

Summary The union-find data structure introduced above allows us to execute

- find(x) in $O(\log(n))$ time
- union(A, B) in$O(1)$ time
- makeUnionFind(S) in $O(n)$ time

if we are dealing with a set S, $|S| = n$, and two subsets $A, B \subseteq S$.

Definition For every node in a union-find data structure we can also assign a *rank* which corresponds to the depth of the subtree rooted at that node.

Note A leaf node has rank 0.

4.4 Implementing Kruskal's algorithm using a union-find data structure

Reminder Kruskal's algorithm identifies a minimum spanning tree (see 3.2.3)

- start with a subgraph G' which contains only the isolated nodes from G (i.e. no edges)
- while the number of edges in G' is $< |V| - 1\{$
  - of all the edges in G that have not yet been added to G', pick the edge e with the lowest $l(e)$
  - add this edge to G' unless it creates a cycle in G'

  $\}$

Implementation Suppose graph G has n nodes and m edges

1. • Sort the edges in G by cost $l(e) \rightarrow$ requires $O(m \log(m))$ time
   • As we have at most one edge between any possible pairs of nodes, we know that $m < n^2$
   $\rightarrow$ sorting the edges in G by cost $l(e)$ requires $O(m \log(n))$ time
2. • we store each of the connected components of the emerging graph G' in a union-find data structure.
   • when an edge $e = (v, w)$ is considered to be added to G', we computer find(v) and find(w) and test if find(v) = find(w) in order to determine if nodes v and w belong to different components
   • if edge e is included in G', we merge the corresponding two components using union(find(v), find(w))
3. • throughout the algorithm we are executing at most
   – $2m$ find operations at a cost of $O(\log(n)$ each
   – $(n-1)$ union operations at a cost of $O(1)$ each
   overall this requires $O(m \log(n))$ time

Conclusion Using a union-find data structure and an input graph G with n nodes and m edges, Kruskal's algorithm can be implemented to run in $O(m \log(n))$ time.

Further Implementation Improvements: *Path Compression*

- assume that v is a node for which find(v) takes $\log(n)$ time
- realize: after the first execution of find(v), we already know the name x of the set containing v and the same holds for all nodes that we encounter on our path to the root node

Idea
- after each find(v) call, reset all pointers along the path from v to the root node to point directly to the root node
- result: all subsequent find calls to any node on a previously encountered find path are faster
- By bounding the total time of a sequence of n find operations rather than the worst-case time for any one of them separately, we can show that n find operations take

$$O(n\alpha(n))\text{time}$$

where α is the *inverse Achermann function* which is such a slow-growing function of n that $n\alpha(n)$ is almost linear (and $\alpha(n) \leq 4$ for values of n that are typically encountered in practice)

Note: Using path compression does *not* lower the time requirements for Kruskal's algorithm as we will still need $O(m \log(n)$ time for the initial sorting of edges.