Minimum Spanning Trees

Kruskal's MST Algorithm

Algorithms: Design and Analysis, Part II

MST Review

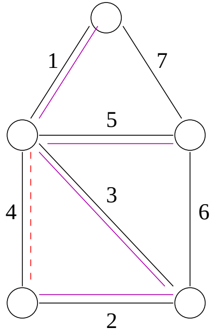
Input: Undirected graph $G = (V, E)$, edge costs c_e.

Output: Min-cost spanning tree (no cycles, connected).

Assumptions: G is connected, distinct edge costs.

Cut Property: If e is the cheapest edge crossing some cut (A, B), then e belongs to the MST.

Example



Kruskal's MST Algorithm

- Sort edges in order of increasing cost

[Rename edges $1, 2, \ldots, m$ so that $c_1 < c_2 < \ldots < c_m$]

- $T = \emptyset$
- For $i = 1$ to m
  - If $T \cup \{i\}$ has no cycles
    - Add i to T
- Return T

Minimum Spanning Trees

Proof of the Cut Property

Algorithms: Design and Analysis, Part II

The Cut Property

Assumption: Distinct edge costs.

CUT PROPERTY: Consider an edge e of G. Suppose there is a cut (A, B) such that e is the cheapest edge of G that crosses it. Then e belongs to the MST of G.

Proof Plan

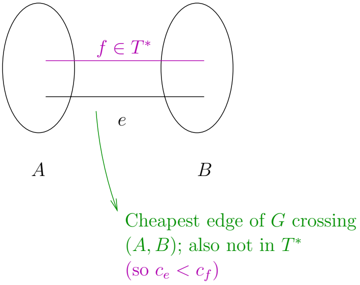
Will argue by contradiction, using an exchange argument.
[Compare to scheduling application]

Suppose there is an edge e that is the cheapest one crossing a cut (A, B), yet e is not in the MST T^*.

Idea: Exchange e with another edge in T^* to make it even cheaper (contradiction).

Question: Which edge to exchange e with?

Attempted Exchange



Note: Since T^* is connected, must construct an edge $f (\neq e)$ crossing (A, B).

Idea: Exchange e and f to get a spanning tree cheaper than T^* (contradiction).

Exchanging Edges

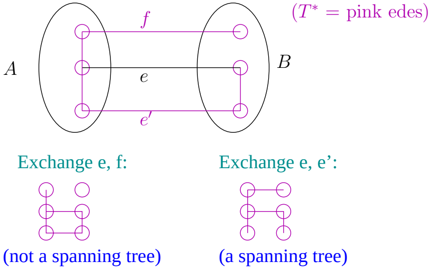
Question: Let T^* be a spanning tree of G, $e \notin T^*$, $f \in T^*$. Is $T^* \cup \{e\} - \{f\}$ a spanning tree of G?

A) Yes always

B) No never

C) If e is the cheapest edge crossing some cut, then yes

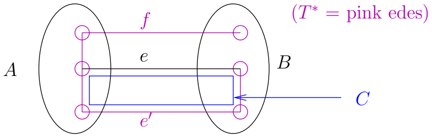
D) Maybe, maybe not (depending on the choice of e and f)



Smart Exchanges

Hope: Can always find suitable edge e' so that exchange yields bona fide spanning tree of G.

How? Let C = cycle created by adding e to T^*.



By the Double-Crossing Lemma: Some other edge e' of C [with $e' \neq e$ and $e' \in T^*$] crosses (A, B).

You check: $T = T^* \cup \{e\} - \{e'\}$ is also a spanning tree.

Since $c_e < c_{e'}$, T cheaper than purported MST T^*, contradiction.

Minimum Spanning Trees

Algorithms: Design and Analysis, Part II

Correctness of Kruskal's Algorithm

Correctness of Kruskal (Part I)

Theorem: Kruskal's algorithm is correct.

Proof: Let $T^* =$ output of Kruskal's algorithm on input graph G.

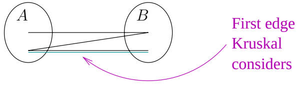
(1) Clearly T^* has no cycles.

(2) T^* is connected. Why?

(2a) By Empty Cut Lemma, only need to show that T^* crosses every cut.

(2b) Fix a cut (A, B). Since G connected at least one of its edges crosses (A, B).

Key point: Kruskal will include first edge crossing (A, B) that it sees [by Lonely Cut Corollary, cannot create a cycle]



Correctness of Kruskal (Part II)

(3) Every edge of T^* satisfied by the Cut Property. (Implies T^* is the MST)

Reason for (3): Consider iteration where edge (u, v) added to current set T. Since $T \cup \{(u, v)\}$ has no cycle, T has no $u - v$ path.

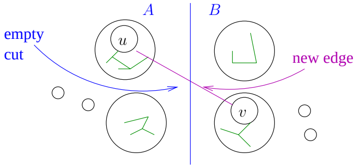
$\Rightarrow \exists$ empty cut (A, B) separating u and v. (As in proof of Empty Cut Lemma)

$\Rightarrow$ By (2b), no edges crossing (A, B) were previously considered by Kruskal's algorithm.

$\Rightarrow (u, v)$ is the first (+ hence the cheapest!) edge crossing (A, B).

$\Rightarrow (u, v)$ justified by the Cut Property. QED

Current State:



Minimum Spanning Trees

Algorithms: Design and Analysis, Part II

Implementing Kruskal's Algorithm via Union-Find

Kruskal's MST Algorithm

- Sort edges in order of increasing cost. ($O(m \log n)$, recall $m = O(n^2)$ assuming nonparallel edges)
- $T = \emptyset$
  - For $i = 1$ to m ($O(m)$ iterations)
    - If $T \cup \{i\}$ has no cycles ($O(n)$ time to check for cycle [Use BFS or DFS in the graph (V, T) which contains $\leq n-1$ edges])
      - Add i to T
- Return T

Running time of straightforward implementation: ($m = \#$ of edges, $n = \#$ of vertices) $O(m \log n) + O(mn) = O(mn)$

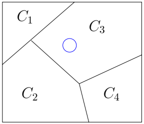
Plan: Data structure for $O(1)$-time cycle checks $\Rightarrow O(m \log n)$ time.

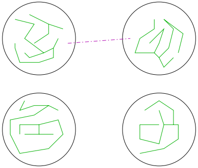
The Union-Find Data Structure

Raison d'être of union-find data structure: Maintain partition of a set of objects.

FIND(X): Return name of group that X belongs to.

UNION(C_i, C_j): Fuse groups C_i, C_j into a single one.





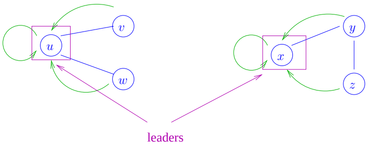
Why useful for Kruskal's algorithm: Objects = vertices

- Groups = Connected components w.r.t. chosen edges T.
- Adding new edge (u, v) to T $\iff$ Fusing connected components of u, v.

Union-Find Basics

Motivation: $O(1)$-time cycle checks in Kruskal's algorithm.

Idea #1: - Maintain one linked structure per connected component of (V, T).
- Each component has an arbitrary leader vertex.



Invariant: Each vertex points to the leader of its component ["name" of a component inherited from leader vertex]

Key point: Given edge (u, v), can check if u & v already in same component in $O(1)$ time. [if and only if leader pointers of u, v match, i.e., FIND(u)=FIND(v)] $\Rightarrow$ $O(1)$-time cycle checks!

Tim Roughgarden

Maintaining the Invariant

Note: When new edge (u, v) added to T, connected components of u & v merge.

Question: How many leader pointer updates are needed to restore the invariant in the worst case?

A) $\Theta(1)$

B) $\Theta(\log n)$

C) $\boxed{\Theta(n)}$ (e.g., when merging two components with $n/2$ vertices each)

D) $\Theta(m)$

Maintaining the Invariant (con'd)

Idea #2: When two components merge, have smaller one inherit the leader of the larger one. [Easy to maintain a size field in each component to facilitate this]

Question: How many leader pointer updates are now required to restore the invariant in the worst case?

A) $\Theta(1)$

B) $\Theta(\log n)$

C) $\Theta(n)$ (for same reason as before, i.e., when merging two components with $n/2$ vertices each)

D) $\Theta(m)$

Updating Leader Pointers

But: How many times does a single vertex v have its leader pointer updated over the course of Kruskal's algorithm?

A) $\Theta(1)$

B) $\boxed{\Theta(\log n)}$

C) $\Theta(n)$

D) $\Theta(m)$

Reason: Every time v's leader pointer gets updated, population of its component at least doubles $\Rightarrow$ Can only happen $\leq \log_2 n$ times.

Running Time of Fast Implementation

Scorecard:

$O(m \log n)$ time for sorting

$O(m)$ times for cycle checks [$O(1)$ per iteration]

$O(n \log n)$ time overall for leader pointer updates

$O(m \log n)$ total (Matching Prim's algorithm)

Minimum Spanning Trees

State-of-the-Art and Open Questions

Algorithms: Design and Analysis, Part II

State-of-the-Art MST Algorithms

Question: Can we do better than $O(m \log n)$? (Running time of Prim/Kruskal.)

Answer: Yes!

$O(m)$ randomized algorithm [Karger-Klein-Tarjan JACM 1995]

$O(m\,\alpha(n))$ deterministic [Chazelle JACM 2000]

"Inverse Ackerman Function": In particular, grows much slower than $\log^* n :=$ # of times you can apply log to n until result drops below 1 (inverse of "tower function" $2^{2^{2^{\cdots^{2}}}}$)

Open Questions

Weirdest of all: [Pettie/Ramachandran JACM 2002] Optimal deterministic MST algorithm, but precise asymptotic running time is unknown! [Between $\Theta(m)$ and $\Theta(m\alpha(n))$, but don't know where]

Open Questions:
- Simple randomized $O(m)$-time algorithm for MST [Sufficient: Do this just for the "MST verification" problem]
- Is there a deterministic $O(m)$-time algorithm?

Further reading: [Eisner 97]

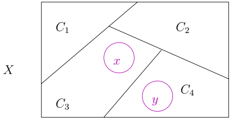
Advanced Union-Find

Lazy Unions

Algorithms: Design and Analysis, Part II

The Union-Find Data Structure

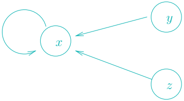
Raison d'être: Maintain a partition of a set X.



FIND: Given $x \in X$, return name of x's group.
UNION: Given x & y, merge groups containing them.

Previous solution (for Kruskal's MST algorithm)

- Each $x \in X$ points directly to the "leader" of its group.

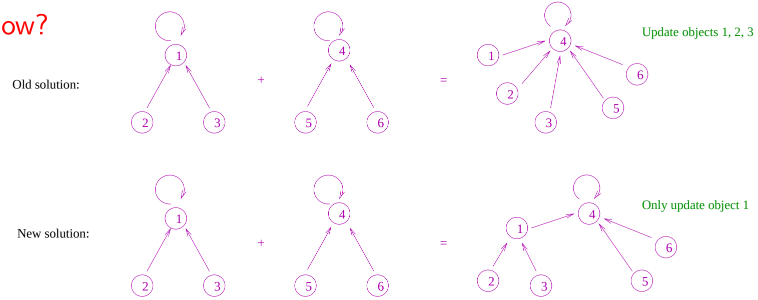


- $O(1)$ FIND [just return x's leader]
- $O(n \log n)$ total work for n UNIONS [when 2 groups merge, smaller group inherits leader of larger one]

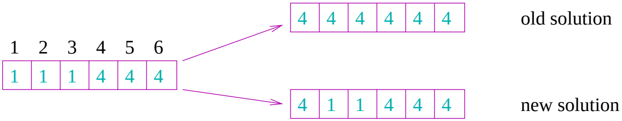
Lazy Unions

New idea: Update only <u>one</u> pointer each merge!

How?

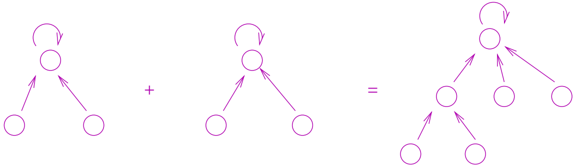


In array representation: (Where $A[i] \leftrightarrow$ name of i's parent)



How to Merge?

In general: When two groups merge in a UNION, make one group's leader (i.e., root of the tree) a child of the other one.



Pro: UNION reduces to 2 FINDS [r_1=FIND(x), r_2=FIND(y)] and $O(1)$ extra work [link r_1, r_2 together]

Con: To recover leader of an object, need to follow a path of parent pointers [not just one!]
$\Rightarrow$ Not clear if FIND still takes $O(1)$ time.

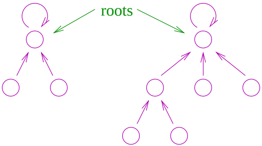
Advanced Union-Find

Union by Rank

Algorithms: Design and Analysis, Part II

The Lazy Union Implementation

New implementation: Each object $x \in X$ has a <u>parent</u> field.



Invariant: Parent pointers induce a collection of directed trees on X. (x is a root $\iff$ parent[x]=x)

Initially: For all x, parent[x]=x

FIND(x): Traverse parent pointers from x until you hit the root.

UNION(x, y): s_1 =FIND(x); s_2 =FIND(y); Reset parent of one of s_1, s_2 to be the other.

Quiz on Lazy Unions

Question: Suppose, in the UNION operation, we choose the new root arbitrarily from the two old ones. What is the worst-case running time of the FIND and UNION operations, respectively?

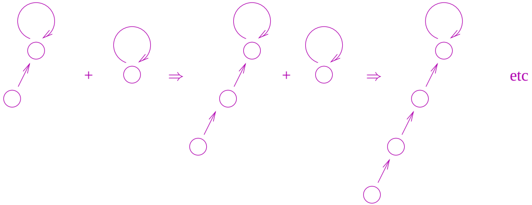
A) $\Theta(1), \Theta(1)$

B) $\Theta(\log n), \Theta(1)$

C) $\Theta(\log n), \Theta(\log n)$

D) $\boxed{\Theta(n), \Theta(n)}$

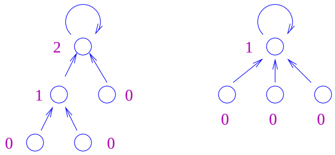
Issue: Scraggly trees:



Union by Rank

Ranks: For each $x \in X$, maintain field rank[x].

[In general rank[x]=1+(max rank of x's children)]



Invariant (for now): For all $x \in X$, rank[x]=maximum number of hops from some leaf to x.

[Initially, rank[x]=0 for all $x \in X$]

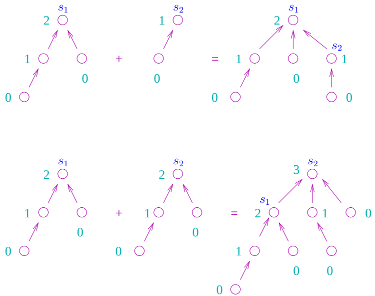
To avoid scraggly trees ("Union by Rank"): Given x & y:

- s_1=FIND(x), s_2=FIND(y)
- If rank[s_1]>rank[s_2] then set parent[s_2] to s_1 else set parent[s_1] to s_2.

To-do: Update ranks to restore Invariant.

Quiz on Rank Updates

Question: Recall s_1=FIND(x), s_2=FIND(y). How do the ranks of s_1 & s_2 change after UNION(x, y)?



A) Unchanged

B) The one with larger rank goes up by 1

C) The one with smaller rank goes up by 1

D) No change unless ranks of s_1, s_2 were equal, in which case s_2's rank goes up by 1

Advanced Union-Find

Union by Rank - Analysis

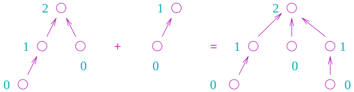
Algorithms: Design and Analysis, Part II

Properties of Ranks

Recall: Lazy Unions.

Invariant (for now): rank[x] = max # of hops from a leaf to x.
[Note $\max_x$rank[x] $\approx$ worst-case running time of FIND.]

Union by Rank: Make old root with smaller rank child of the root with the larger rank.
[Choose new root arbitrarily in case of a tie, and add 1 to its rank.]



Immediate from Invariant/Rank Maintenance:

(1) For all objects x, rank[x] only goes up over time
(2) Only ranks of roots can go up
[once x a non-root, rank[x] frozen forevermore]
(3) Ranks strictly increase along a path to the root

Rank Lemma

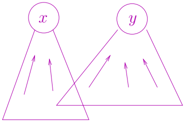
Rank Lemma: Consider an arbitrary sequence of UNION (+FIND) operations. For every $r \in \{0, 1, 2, \ldots\}$, there are at most $n/2^r$ objects with rank r.

Corollary: Max rank always $\leq \log_2 n$

Corollary: Worst-case running time of FIND, UNION is $O(\log n)$.
[With Union by Rank.]

Proof of Rank Lemma

Claim 1: If x, y have the same rank r, then their subtrees (objects from which can reach x, y) are disjoint.



Claim 2: The subtree of a rank-r object has size $\geq 2^r$.
[Note Claim 1 + Claim 2 imply the Rank Lemma.]

Proof of Claim 1: Will show contrapositive. Suppose subtrees of x, y have object z in common $\Rightarrow \exists$ paths $z \rightarrow x, z \rightarrow y$
$\Rightarrow$ One of x, y is an ancestor of the other
$\Rightarrow$ The ancestor has strictly larger rank. [By property (3)]
QED (Claim 1)

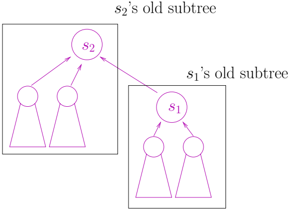
Proof of Claim 2

Rank $r \Rightarrow$ Subtree size $\geq 2^r$

Base case: Initially all ranks $= 0$, all subtree sizes $= 1$

Inductive step: Nothing to prove unless the rank of some object changes (subtree sizes only go up).

Interesting case: UNION(x, y), with s_1=FIND(x), s_2=FIND(y), and rank$[s_1]$=rank$[s_2]$=$r \Rightarrow s_2$'s new rank $= r + 1$
$\Rightarrow s_2$'s new subtree size $= s_2$'s old subtree size $+ s_1$'s old subtree size (each at least 2^r by the inductive hypothesis) $\geq 2^{r+1}$. QED!



Advanced Union-Find

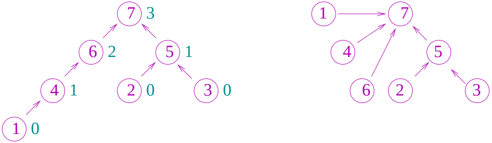
Path Compression

Algorithms: Design and Analysis, Part II

Path Compression

Idea: Why bother traversing a leaf-root path multiple times?

Path compression: After FIND(x), install shortcuts (i.e., revise parent pointers) to x's root all along the $x \rightarrow$ root path.



In array representation:

1	2	3	4	5	6	7
4	5	5	6	7	7	7

$\longrightarrow$

1	2	3	4	5	6	7
7	5	5	7	7	7	7

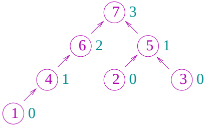
Con: Constant-factor overhead to FIND (from "multitasking").

Pro: Speeds up subsequent FINDs. [But by how much?]

On Ranks

Important: Maintain all rank fields EXACTLY as without path compression.

- Ranks initially all 0
- In UNION, new root = old root with bigger rank
- When merging two nodes of common rank r, reset new root's rank to $r + 1$



Bad news: Now rank[x] is only an upper bound on the maximum number of hops on a path from a leaf to x
(which could be much less)

Good news: Rank Lemma still holds ($\leq n/2^r$ objects with rank r)

Also: Still always have rank[parent[x]]$>$rank[x] for all non-roots x

Hopcroft-Ullman Theorem

Theorem: [Hopcroft-Ullman 73] With Union by Rank and path compression, m Union+Find operations take $O(m \log^* n)$ time, where $\log^* n =$ the number of times you need to apply log to n before the result is ≤ 1.

Quiz on $\log^*$

Question: What is $\log^*(2^{65536})$?

A) 2
B) $\boxed{5}$
C) 16
D) 65536

In general: $\log^*(2^{2^{\cdots \ t \text{ times } \cdots^{2}}}) = t$

Measuring Progress

Advanced Union-Find

Path Compression: The Hopcroft-Ullman Analysis

Algorithms: Design and Analysis, Part II

Hopcroft-Ullman Theorem

Theorem: [Hopcroft-Ullman 73] With Union by Rank and path compression, m UNION+FIND operations take $O(m \log^* n)$ time, where $\log^* n =$ the number of times you need to apply log to n before the result is ≤ 1.

[Will focus on interesting case where $m = \Omega(n)$]

Measuring Progress

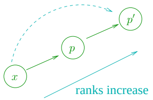
Intuition: Installing shortcuts should significantly speed up subsequent FINDs+UNIONs.

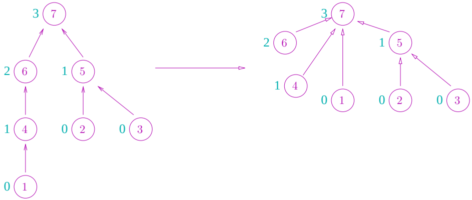
Question: How to track this progress and quantify the benefit?

Idea: Consider a non-root object x. $\longrightarrow$ Recall: rank[x] frozen

Progress measure: rank[parent[x]] - rank[x]

Path compression increases this progress measure: If x has old parent p, new parent $p' \neq p$, then rank[p']>rank[p].





Proof Setup

Rank blocks: $\{0\}, \{1\}, \{2, 3, 4\}, \{5, \ldots, \overset{16}{2^4}\}, \{17, 18, \ldots, \overset{65536}{2^{16}}\}$, $\{65537, \ldots, 2^{65536}\}, \ldots, \{\ldots, n\}$

Note: There are $O(\log^* n)$ different rank blocks.

Semantics: Traversal $x \rightarrow$ parent(x) is "fast progress" $\iff$ rank[parent[x]] in larger block than rank[x]

Definition: At a given point in time, call object x <u>good</u> if

(1) x or x's parent is a root OR

(2) rank[parent[x]] in larger block than rank[x]

x is <u>bad</u> otherwise.

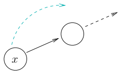
Proof of Hopcroft-Ullman

Point: Every FIND visits only $O(\log^* n)$ good nodes [$2 + \#$ of rank blocks $= O(\log^* n)$]

Upshot: Total work done during m operations $= O(m \log^* n)$ (visits to good objects) $+$ total $\#$ of visits to bad nodes (need to bound globally by separate argument)

Consider: A rank block $\{k+1, k+2, \ldots, 2^k\}$.

Note: When a bad node is visited



its parent is changed to one with strictly larger rank $\Rightarrow$ Can only happen 2^k times before x becomes good (forevermore).

Proof of Hopcroft-Ullman II

Total work: $O(m \log^* n) + O(\boxed{\# \text{ visits to bad nodes}})$.

$\leq n$ for each of $O(\log^* n)$ rank blocks

Consider: A rank block $\{k+1, k+2, \ldots, 2^k\}$.

Last slide: For each object x with final rank in this block, # visits to x while x is bad is $\boxed{\leq 2^k.}$

Rank Lemma: Total number of objects x with final rank in this rank block is $\sum_{i=k+1}^{2^k} n/2^i \boxed{\leq n/2^k.}$

$\leq n$ visits to bad objects in this rank block.

Recall: Only $O(\log^* n)$ rank blocks.

Total work: $O((m+n)\log^* n)$.

Advanced Union-Find

The Ackermann Function

Algorithms: Design and Analysis, Part II

Tarjan's Bound

Theorem: [Tarjan 75] With Union by Rank and path compression, m UNION+FIND operations take $O(m\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function (will define in this video)

Proof in next video.

The Ackermann Function

Aside: Many different definitions, all more or less equivalent.

Will define $A_k(r)$ for all integers k and $r \geq 1$. (recursively)

Base case: $A_0(r) = r + 1$ for all $r \geq 1$.

In general: For $k, r \geq 1$:

$$\begin{aligned} A_k(r) &= \text{Apply } A_{k-1} \ r \text{ times to } r \\ &= \boxed{(A_{k-1} \circ A_{k-1} \circ \ldots \circ A_{k-1})(r)} \end{aligned}$$

r-fold composition

Quiz: A_1

Quiz: $A_1(r)$ corresponds to what function of r?

A) Successor ($r \mapsto r + 1$)

B) $\boxed{\text{Doubling } (r \mapsto 2r)}$

C) Exponentiation ($r \mapsto 2^r$)

D) Tower function ($r \mapsto 2^{2^{\cdots\ r \text{ times } \cdots^{2}}}$)

$A_1(r) = (A_0 \circ A_0 \circ \ldots \circ A_0)(r) = 2r$

(r-fold composition, add 1 each time)

Quiz: A_2

Quiz: What function does $A_2(r)$ correspond to?

A) $r \mapsto 4r$

B) $r \mapsto 2^r$

B) $\boxed{r \mapsto r2^r}$

D) $r \mapsto 2^{2^{\cdots \text{ } r \text{ times } \cdots^{2}}}$

$A_2(r) = (A_1 \circ A_1 \circ \ldots \circ A_1)(r) = r2^r$

(r-fold composition, doubles each time)

Quiz: A_3

Quiz: What is $A_3(2)$? Recall $A_2(r) = r2^r$

A) 8

B) 1024

B) $\boxed{2048}$

D) Bigger than 2048

$A_3(2) = A_2(A_2(2)) = A_2(8) = 82^8 = 2^{11} = 2048$

In general: $A_3(r) = (A_2 \circ A_2 \circ \ldots (r \text{ times}) \ldots \circ A_2)(r) \geq$ a tower of r 2's $= 2^{2^{\cdots \, r \text{ times} \, \cdots^{2}}}$

A_4

$A_4(2) = A_3(A_3(2)) = A_3(2048) \geq 2^{2^{\cdots \text{ height } 2048 \ \cdots^{2}}}$

In general: $A_4(r) = (A_3 \circ \ldots \ r \text{ times } \ldots \circ A_3)(r) \approx$ iterated tower function (aka "wowzer" function)

The Inverse Ackermann Function

Definition: For every $n \geq 4$, $\alpha(n) =$ minimum value of k such that $A_k(2) \geq n$.

$\alpha(n) = 1$, $n = 4$ $(A_1(2) = 4)$
$\alpha(n) = 2$, $n = 5, \ldots, 8$ $(A_2(2) = 8)$
$\alpha(n) = 3$, $n = 9, 10, \ldots, 2048$
$\alpha(n) = 4$, n up to roughly a tower of 2's of height 2048
$\alpha(n) = 5$ for n up to ???

$\log^* n = 1$, $n = \underline{2}$
$\log^* n = 2$, $n = 3, \underline{4}$
$\log^* n = 3$, $n = 5, \ldots, \underline{16}$
$\log^* n = 4$, $n = 17, \underline{65536}$
$\log^* n = 5$, $n = 65537, \underline{2^{65536}}$
$\log^* n = 2048$ for such n

Advanced Union-Find

Tarjan's Analysis

Algorithms: Design and Analysis, Part II

Tarjan's Bound

Theorem: [Tarjan 75] With Union by Rank and path compression, m UNION+FIND operations take $O(m\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function

Acknowledgement: Kozen, "Design and Analysis of Algorithms"

Building Blocks of Hopcroft-Ullman Analysis

Block #1: Rank Lemma (at most $n/2^r$ objects of rank r)

Block #2: Path compression $\Rightarrow$ If x's parent pointer updated from p to p', then rank(p')$\geq$rank(p)$+1$

New idea: Stronger version of building block #2. In most cases, rank of new parent <u>much</u> bigger than rank of old parent (not just by 1).

Quantifying Rank Gaps

Definition: Consider a non-root object x (so rank[x] fixed forevermore)

Define $\delta(x)$ = max value of k such that rank[parent[x]$\geq A_k$(rank[x])

(Note $\delta(x)$ only goes up over time)

Examples: Always have $\delta(x) \geq 0$

$\delta(x) \geq 1 \iff$ rank[parent[x]]$\geq$ 2 rank[x]

$\delta(x) \geq 2 \iff$ rank[parent[x]]$\geq$ rank[x] $2^{\mathrm{rank}[x]}$

Note: For all objects x with rank[x]≥ 2, then $\delta(x) \leq \alpha(n)$
[Since $A_{\alpha(n)}(2) \geq n$]

Good and Bad Objects

Definition: An object x is bad if all of the following hold:

(1) x is not a root

(2) parent(x) is not a root

(3) rank(x)≥ 2

(4) x has an ancestor y with $\delta(y) = \delta(x)$

x is good otherwise.

Quiz

Question: What is the maximum number of good objects on an object-root path?

A) $\Theta(1)$

B) $\boxed{\Theta(\alpha(n))}$

C) $\Theta(\log^* n)$

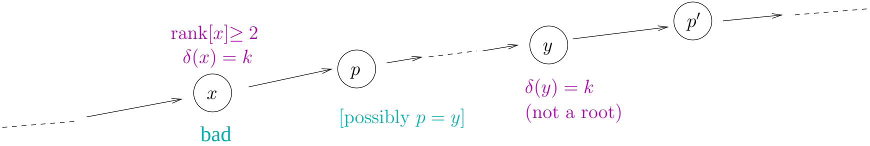
D) $\Theta(\log n)$

$\leq$ 1 root + 1 child of root
+ 1 object with rank 0
+ 1 object with rank 1
+ 1 object with $\delta(x) = k$
for each $k = 0, 1, 2, \ldots, \alpha(n)$

Proof of Tarjan's Bound

Upshot: Total work of m operations $= O(m\alpha(n))$ (visits to good objects)+ total # of visits to bad objects (will show is $O(n\alpha(n))$)

Main argument: Suppose a FIND operation visits a bad object x:



Path compression: x's new parent will be p' or even higher.

$\Rightarrow$ rank[x's new parent] $\geq$ rank[p'] $\geq$ A_k(rank[y]) $\geq$ A_k(rank[p])

ranks only go up   since $\delta(y) = k$ ranks only go up

Proof of Tarjan's Bound II

Point: Path compression (at least) applies the A_k function to rank[x's parent]

Consequence: If r=rank[x] (≥ 2), then after r such pointer updates we have

$$\text{rank}[x\text{'s parent}] \geq (A_k \circ \ldots \text{ r times } \ldots \circ A_k)(r) = A_{k+1}(r)$$

Definition of Ackermann function

Thus: While x is bad, every r visits increases $\delta(x)$
$\Rightarrow \leq r\alpha(n)$ visits to x while it's bad

Proof of Tarjan's Bound III

Recall: Total work of m operations is $O(m\alpha(n))$ (visits to good objects) $+$ total # of visits to bad objects.

$\leq \sum_{\text{objects } x} rank[x]\alpha(n)$

$= \alpha(n) \sum_{r \geq 0} r \;$ (# of objects with rank r)

$\leq n/2^r$ for each r by the Rank Lemma

$= n\alpha(n) \sum_{r \geq 0} r/2^r \longrightarrow = O(1)$

$= O(n\alpha(n))$. QED!

Epilogue

"This is probably the first and maybe the only existing example of a simple algorithm with a very complicated running time. . . . I conjecture that there is no linear-time method, and that the algorithm considered here is optimal to within a constant factor."

-Tarjan, "Efficiency of a Good But Non Linear Set Union Algorithm", Journal of the ACM, 1975.

Conjecture proved by [Fredman/Saks 89]!