Week 5

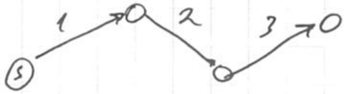
Dijkstra's Shortest Path Algorithm

problem: Single-Source Shortest Paths

input: directed graph $G=(V,E)$,
$(m=|E|, n=|V|)$

- each edge has non-negative length l_e
- source vertex s

output: for each $v \in V$, compute
$L(v)$ = length of a shortest s-v path in G



len = 1+2+3 = 6

assumptions

- $\forall v \in V$ $\exists s \to v$ path (for convenience)
- $l_e \geq 0$, $\forall e \in E$ (non negative weights!)

Dijkstra's algorithm can't hande them

BFS computes the shortest path, but only when edges have weight = 1.

Dijkstra's Algorithm

close cousin of BFS

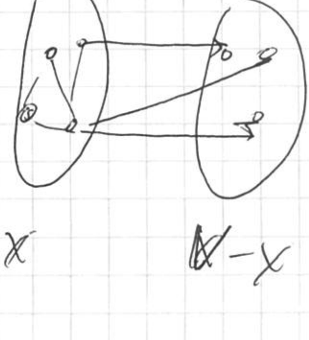
Initialize:

- $X = [s]$ (vertices we've processed so far)
- $A[s] = 0$ (shortest path distance)
  at the end of the algorithm it'll be populated with shortest paths
- $B[s]$ = empty path (computed shortest path)
  (for explanation only)

Main Loop:

while $X \neq V$

we examine all edges that come from X to $V-X$



and among all verticies we pick one which gives the minimal score

- among all edges (~~v,w~~ $(v,w) \in E$ with $v \in X$, $w \notin X$, pick the one that minimizes

$$A[v] + l_{vw} \quad \left[\begin{array}{c}\text{Dijkstra's} \\ \text{greedy} \\ \text{criterion}\end{array}\right]$$

already computed in earlier iteration

we will call the minimizing edge (v^*, w^*)

- add w^* to X

- $A[w^*] = A[v] + l_{v^*w^*}$

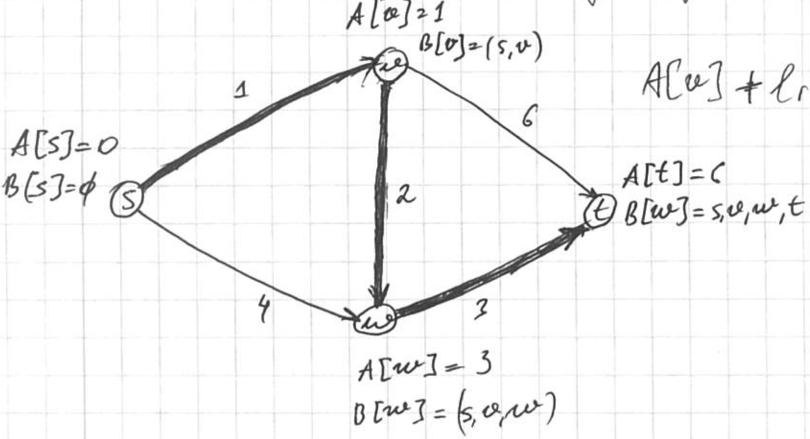
shortest path from s to w^*

- $B[w^*] = B[v^*] \cdot u(v^*, w^*)$

Example

greedy score for (v,w

$A[v] + l_{vw}$

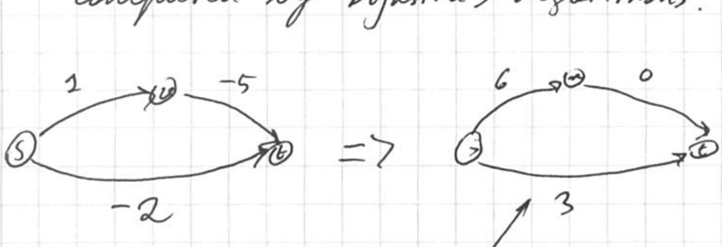


Non-example

Question: Why not reduce computing shortest path with negative edge lengths ~~the~~ to the same problem with non-negative lengths?
(by adding large constant to edge lengths?

It doesn't preserve the shortest path! →

and anyway the shortest path won't be computed by Dijkstra's algorithms.



$s \to v \to t$ shortest

becomes the shortest

Correctness

Theorem: For every directed graph with non negative edge lengths, the algo. computes all shortest path distances

$$\text{i.e. } \underbrace{A[v]}_{\text{what algorithm computes}} = \underbrace{L(v)}_{\text{true shortest distance from } s \text{ to } v} \quad \forall v \in V$$

proof: by induction.

base case:

$$A[s] = L[s] = 0 \quad \text{(correct)}$$

hypothesis:

$A[v] = L[v]$, $B[v]$ – true shortest path

In current iteration

we pick an edge $(\overset{\in X}{v^*}, \overset{\notin X}{w^*})$ and we add w^* to X

we set $\underbrace{B[w^*]}_{\text{length } L[v^*] + l_{v^*w^*}} = \underbrace{B[v^*]}_{\text{shortest path}} \cup (v^*, w^*)$

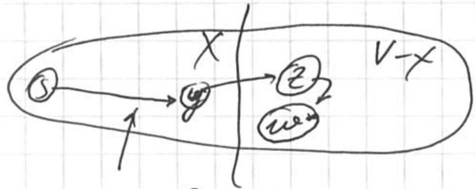
has length $L(v^*)$

We need to show that every s-w^* path has lenght $\geq L(v^*) + l_{v^*w^*}$

(if so, our path is shortest)

Let P = any $s \to w^*$ path

$s \in X$, $w^* \notin X$ $\Rightarrow$ Must cross the frontier



a path P

P: 

length = l_{yz}

$\geq$ len of the shortest path s-y

$L(y) = A[y]$

(by inductive hypothesis)

$(y \in X)$

<u>So</u>: the total len P at least

$$A[y] + l_{yz} \quad \begin{pmatrix} y \in X \\ z \notin X \end{pmatrix}$$

by Dijkstra's greedy <u>criterion</u>:

$$\overbrace{A[v^*] + l_{v^*w^*}}^{\text{our path}} \leq A[y] + l_{yz} \leq \text{len of } P$$

Q.E.D.

Implementation

don't need the B array

m - number of edges

n - vertices (nodes)

$\Theta(mn)$ - naïve implementation of Dijkstra's

- $(n-1)$ iteration of while loop
- $\Theta(m)$ work per iteration
- $\Theta(1)$ work per edge

Heap Operations.

we're asking for the minimum over and over again!

key properties:

- at every node, key $\leq$ children's keys
- balanced tree

- extract-min by swapping up last leaf, bubbling down

- insert via bubbling up

- height $\approx \log_2 n$

Operations: all in $O(\log_2 n)$ time

Invariants:

#1 elements in heap: vertices in $V-X$

#2 for $v \notin X$

key$[v]$ = smallest Dijkstra's greedy ~~score~~ score of any edge $\underline{(u,v)}$ in E with u in X

$+\infty$ - no such edges exist

So if we maintain these 2 ~~variable~~ invariants,

extract-min yields correct ~~ver~~ vertex w^* to add to X next

(and we set $A[w^*]$ to $\underline{\text{key}}[w^*]$

to maintain the invariants:

#2: $\forall v \in X$

$key[v]$ = smallest Dij's greedy score of edge (u,v) with u in X

When w extracted from heap (i.e. added to X)

- for each edge $(w,v) \in E$

  if $v \in V-X$ (i.e. in heap)

  key update

  delete v from heap

  recompute $key[v] = \min\{key[v], A[w] + l_{wv}\}$

  re-insert v onto heap

Running time: $O(m \log_2(n))$

Data Structures

4 levels of data structures knowledge

level 0 what's this?

level 1 cocktail-party level literacy

level 2 ←

level 3 know the guts

Heap

- a container that have keys

\# operations:

- insert - add new object
- extract-min - min from heap

  (ties are broken arbitrarly)

running times $O(\log n)$

- heapify - initialize time in $O(n)$ time
- delete - $O(\log n)$ time

Application:

- canonical usage: fast way to do repeated minimum computations

Heap Sort.

- event manager – „priority queue" (synonym for heap)

- median maintanance

  - given a sequence of numbers: $x_1 \ldots x_n$, one-by-one

  - at each time step i, the median of $\{x_1 \ldots x_i\}$

  - constraint: $O(\log i)$ at step i

Solution:

maintain 2 heaps

H_{LOW} $\quad$ H_{HIGH}

$\downarrow$ $\quad$ $\downarrow$

extract max $\quad$ extract min

key idea: maintain the invariant that

$\sim i/2$ smallest (largest) elements

are in H_{LOW} (H_{HIGH})

$\Downarrow$

so in 20th step, in H_{LOW} would be 11^{th} order statistics

and in H_{HIGH} – 10^{th} order

while keeping the heaps balanced (having the same amount of items)

- Speeding Up Dijkstra

  naive implementation: $\Theta(n m)$

  with heaps $\Rightarrow$ runtime: $O(m \log n)$

Implementation Details

heap - is a tree, complete, binary, rooted

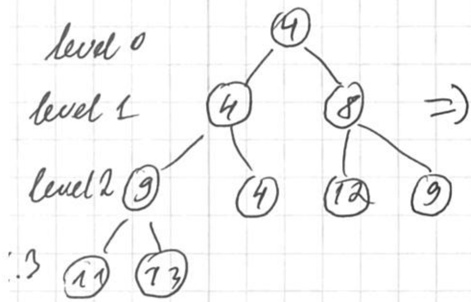
heap property:

at every node x, key $[x] \leq$ all keys of x's children

$\Downarrow$

object at root must have min key value

Array implementation



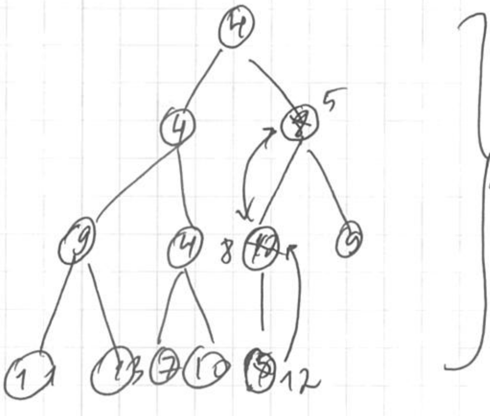
$\Rightarrow$ 4 4 8 9 4 12 9 11 13

$\text{parent}(i) = \lfloor i/2 \rfloor$

round down

children of i: 2i, 2i+1

Insert And Bubble-Up



~ $\log_2 n$ levels

Insert (key k)

- stick k at the end of last level
- bubble-up k until heap property is restored

(i.e. key of k's parent $\leq k$)

~~Delete~~

Extract-Min

1. Delete Root
2. Move last leaf to be new root
3. Iteratively Bubble-Down until heap property has been restored

   [always swap with the smaller child)

Balanced Binary Search Tree

Sorted Arrays

Operations

- Search $\Theta(\log n)$
- Select $O(1)$
- min/max $O(1)$
- pred/succ $O(1)$
- rank $O(\log n)$

- insertion
- deletion

$\left.\right\} \Theta(n)$!

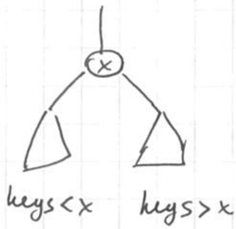
Balanced Trees:

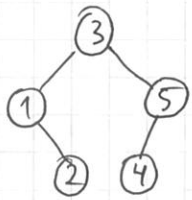
like sorted array + fast (log) inserts and deletes

Binary Search Tree Structure

- has exactly one node per key
- most basic version, each node has
    - left child pointer
    - right child pointer
    - parent pointer

Search tree property $\Rightarrow$
(should hold for every node of the search tree)



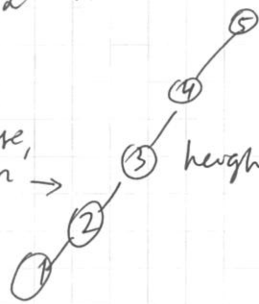


height = 2

the height of a BST

could be anywhere from $\sim \log_2 n$ to $\sim n$

worst case, in a chain



height = 4

△ Searching

- start at the root
- traverse left / right child pointers

  if k < key   if k > key

- return node with key k or null

△ Insert

- search for k (unsuccessfully)
- rewrite final NULL pointer to point to new node with key k

Worst-case running time for Search and Insert –

O(height) (of the tree)

△ Min (max)

start at root and follow left (right) child pointer

△ Pred (prev next smallest element)

- if k's subtree is not empty, return the max key in left subtree
- otherwise follow parent pointer until you get to a key less than k

△ In-Order Traversal

(to print out keys in increasing order)

let r = root of the search tree, TL, TR - subtrees

- recurse on TL
- print out r's key
- recurse on TR

Running time: $O(n)$

△ Deletion

- search for k $\qquad\qquad$ $O(\text{height})$
  - if k has no children
    - just delete the node
  - k has one child

    the child gets the position of k
  - k has two children
    - compute k's predecessor l
      (traverse k's non-NULL left child ptr, then right-child ptr until no longer possible)
    - SWAP k and l
    - in a new position it's easy to delete k
      (k has no right child)

- Select (I want to select an order statistic)
- Rank (how many keys are less or equal to that value?)

Idea: store extra information at each tree node

example: size(x) = number of tree nodes in a subtree rooted at x

$$size(x) = size(l) + size(r) + 1$$

- Select
  (how to select i^{th} order statistic from augmented search trees - with subtree sizes)
  - start at root x with children lt and rt
  - let $a = size(lt)$ [$a = 0$ if lt has no left child]
  - if $a = i-1$, return x's key
  - if $a \geq i$, recursively compute i^{th} order statistic of lt
  - if $a < i-1$, recursively compute $(i-a-1)^{th}$ order statistic of tree rooted at rt.

Running time = Θ(height)

Red-Black trees

Balanced search trees:

idea: the height is always $O(\log n)$ –

all main operations run in $O(\log n)$

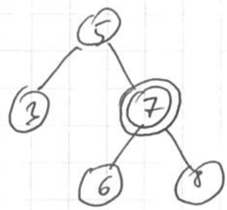
Red-black invariants

- each node red or black
- root is black
- no 2 reds in a row
  (red node $\Rightarrow$ only black children)
- every path from the root to NULL-nodes passes the same amount of black nodes

(unsuccessful search)

Example

◎ - red



Height guarantee

Claim: every red-black tree with n nodes has height $\leq 2 \log_2 (n+1)$