Dijkstra with min-heaps

Monday, 13 September 2021 1:57 PM

DIJKSTRA'S ALGORITHM, MIN HEAPS

Given: directed graph $G = (V, E)$, non negative wt. w_e on each edge $e \in E$, a source vertex s

Problem: Find a min-wt. path from s to all other vertices

(in class, will focus on finding wt. of min-wt. path; extend algo to actually finding the path (by yourself))

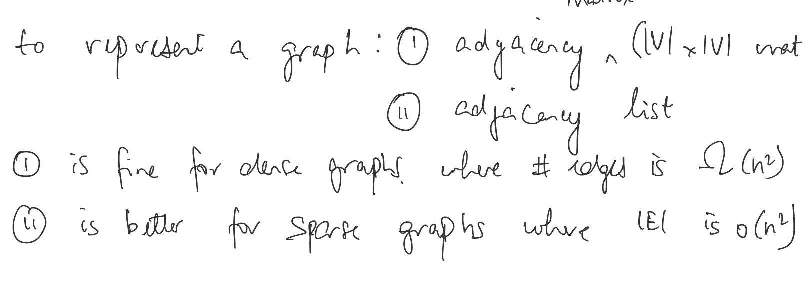
v_3: min wt. $s \to v_3$
wt.: 2

v_1: min wt $s \to v_3 \to v_1$
wt.: 9

to represent a graph: (i) adjacency matrix ($|V| \times |V|$ matrix)
(ii) adjacency list

(i) is fine for dense graphs where # edges is $\Omega(n^2)$
(ii) is better for sparse graphs where $|E|$ is $o(n^2)$

For Dijkstra's algo:



Say this is min-wt $s - v_5$ path.
Then this is also min-wt. path for every vtx. in the path
Note this path has 5 edges. Hence this is also the min-wt. $s - v_5$ path that has ≤ 5 edges
... similarly this path is also the min-wt. $s - v_4$ path that has ≤ 4 edges
Let's say $dist_k(v)$ is the wt. of the min-wt. $s - v$ path that has $\leq k$ edges
Assume we know $dist_{k-1}(v)$ $\forall v \in V$
Then $dist_k(v) = \min_{e=(u,v) \in E} \{ dist_{k-1}(u) + w_e, dist_{k-1}(v) \}$

Algo:

$\forall v \neq s$ $dist_1(v) = \min_{(s,v) \in E} w_{sv}$, $dist_1(s) = 0$

For $k = 2 \dots n-1$ \\ where $n = |V|$
  For all $v \neq s$
    $dist_k(v) = \min_{e=(u,v) \in E} \{ dist_{k-1}(u) + w_e, dist_{k-1}(v) \}$

$dist(v) \leftarrow dist_{n-1}(v)$

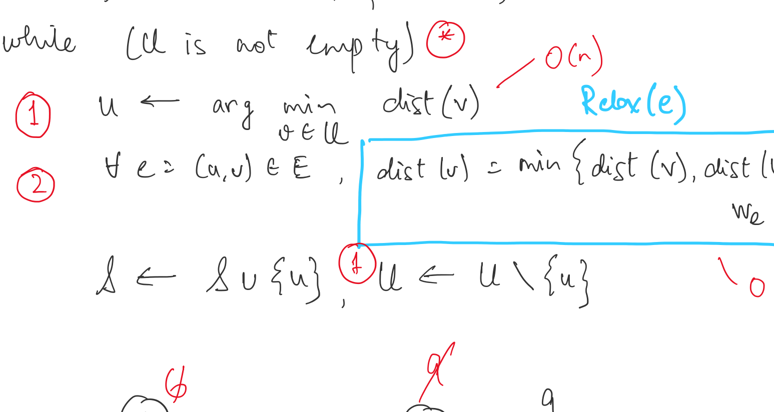
Running Time: naively $n \times |V| \times n = O(n^3)$

Claim: For every $k = 2 \dots n-1$, let $\hat{v}$ be the vtx. that minimizes $dist_k(v)$. Then $dist_k(\hat{v}) = dist_{k+1}(\hat{v}) = \dots = dist_{n-1}(\hat{v}) = dist(\hat{v})$

Using this & some other properties, we get Dijkstra's algorithm:

Dijkstra ($G = (V, E)$, w, s)

$S \leftarrow \phi$ \\ set of vertices for which min-wt. distances known
$U \leftarrow V$ (*)
$dist(v) \leftarrow \infty$ $\forall v \neq s$, $dist(s) \leftarrow 0$
while (U is not empty) (*) — $O(n)$
(1) $u \leftarrow \arg\min_{v \in U} dist(v)$ Relax(e)
(2) $\forall e = (u, v) \in E$, $\boxed{dist(v) = \min \{ dist(v), dist(u) + w_e \}}$ — $O(1)$
$S \leftarrow S \cup \{u\}$, (3) $U \leftarrow U \setminus \{u\}$



$S \leftarrow \phi$, $U \leftarrow V$, $dist(s) = 0$ $dist(v) = \infty$ $\forall v \neq s$
$S \leftarrow \{ s, v_3, v_5, v_4, v_1, v_2 \}$

Need to store: U, $dist(v)$ for all $v \in V$ (but will update it for all $v \in U$)

For data structure, need to be able to:
1. obtain vtx. w/ min. distance in U
2. given a vtx. $v \in U$, decrease distance of v
3. remove vtxs. from U

Idea 1:

Can use an array, each cell of the array has a vtx v & $dist(v)$, extra bit set to 1 if $v \in U$, 0 o.w.

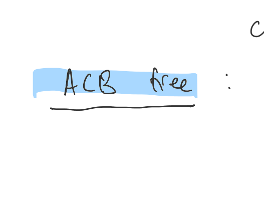
1. $O(n)$
2. $O(1)$ (assume that given a pointer to the vertex in the data structure)
3. $O(m)$

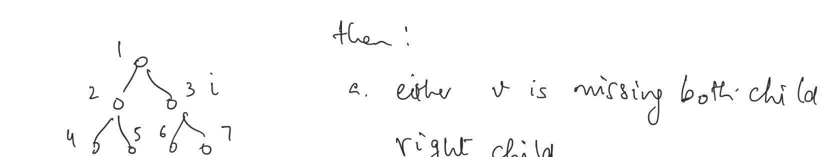
With this data structure algo takes time $O(n^2 + m)$, where $m = |E|$.

Idea 2: Use a min-heap

A data-structure which is an almost complete binary tree, where for every vertex v, value at $v \leq$ value at either of its children

ACB tree: Binary of height h where:
(i) Tree is complete until height $h-1$
(ii) If vtx. v at ht. h is missing a child then:
  a. either v is missing both children or just the right child
  b. all vertices to the right of v at the same level have no children



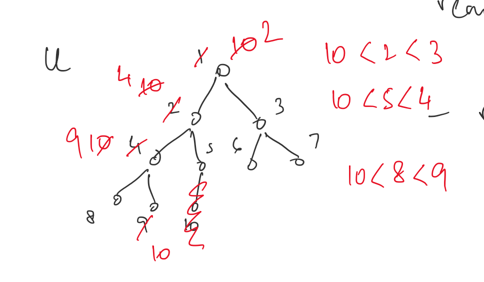


ACB trees of ht. h can be stored as an array of size 2^h where children of vtx. at cell i are in cells $2i, 2i+1$ & for a vtx. at cell i, parent is at $\lfloor i/2 \rfloor$

Now will implement a min-heap to support the above 2 operations.

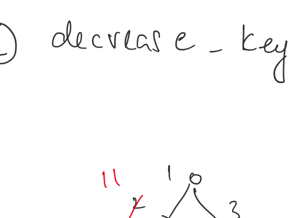
(0) Initialize (V): this creates an array U of size $|V|$ with each entry ∞ (dist of vtx. at that entry)

(1) extract_min (U): return min value in U & also remove this entry from U



remove root, i.e., move last entry to root
(may not be min-heap any more, but is an ACB tree)

(2) decrease_key (i, k) // i is ptr to a vtx. in data structure, k is new value, smaller than old value



Algorithms (assuming min-heap is implemented as an array $U(1) \dots U(t)$)

```
extract_min (U) {
    t ← U.size                // # elements stored in U
    min_val ← U(1).val
    U(1) ← U(t)
    t ← t-1
    i ← 1                     // current position of moved elt.
    while (1) {
        if (2i ≤ t)    left_val ← U(2i).val
        else   return  min_val          // elt. already at leaf
        if (2i+1 ≤ t)  right_val ← U(2i+1).val
        else   right_val ← ∞            // there is no right child
        if (left_val ≤ right_val) && (U(i).val < left_val)
            curr_elt ← U(i)
            U(i) ← U(2i)
            U(2i) ← curr_elt
            i ← 2i
        else if (right_val < left_val) && (U(i).val < right_val)
            curr_elt ← U(i)
            U(i) ← U(2i+1)
            U(2i+1) ← curr_elt
            i ← 2i+1
        else  break
    }
    return  min_val
}

decrease_key (U, i, k) {
    t ← U.size           // # elts. stored in U
    U(i).val ← k         // we assume k ≤ current value at U(i)
    while (i > 1 && U(i).val < U(⌊i/2⌋).val)
        curr_elt ← U(i)
        U(i) ← U(⌊i/2⌋)
        U(⌊i/2⌋) ← curr_elt
        i ← ⌊i/2⌋
}
```

- What is running time of above algorithms?
- prove to yourself that the above algorithms work correctly.