S. KIMMEL

Shortest Paths

← Applications: Linked-In, Bacon#, Maps, Financial

Input: Graph $G=(V,E)$, edge weights ℓ_{vw} for edge (v,w), $s \in V$

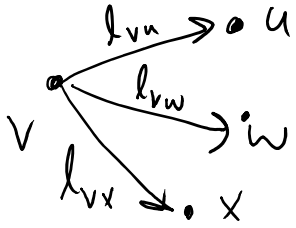
$n=|V|$, $m=|E|$

Output: $\forall\ v \in V$, $\ell(v)$ = shortest path from s to v

$\ell(v)=\infty$ if s, v not connected

Graph given as adjacency list A_G. (array of lists)

$$A_G[v] = \{\{u, \ell_{vu}\}, \{w, \ell_{vw}\}, \{x, \ell_{vx}\}\}$$



BFS (use when all ℓ_{uv} have same value)

```
ℓ[v] = ∞   ∀ v∈V // will store shortest paths
X[v] = False ∀ v∈V   // mark True when "explored", separate Array
A = {}; A.add(s);    //initialize QUEUE
ℓ[s] = 0
X[s] = True
while(A is not empty):
    v = A.pop
    For each edge (v,w):
        If (Ex[w] = false): A.add(w); Ex[v] = true; ℓ[w] = ℓ[v] + 1;
```

← Go to $A_G[v]$ and do For loop through list

S. KIMMEL

Explain: Why is runtime $O(n+m_s)$

- Looks at edges (in for loop)
- Each edge can only be examined when its adjoining vertex is popped. Each vertex can only show up in QUEUE one time. $\Rightarrow$ Each edge is examined once.
- Looking at each edge takes constant time b/c use adjacency list.

$O(m_s)$ to do while loop ← # edges connected to s

+

$O(n)$ to initialize

$=$

$O(n+m_s)$

S. KIMMEL

Runtime of Dijkstra's Algorithm

$X = \{s\}$

$A[s] = 0$

$B[s] = \phi$

While $X \neq V$

- among edges $(v,w) \in E$ with $v \in X$, $w \in V - X$, pick edge that minimizes

$$A[v] + \ell_{vw}$$

⟸ How to do this step

Dijkstra's greedy criterion

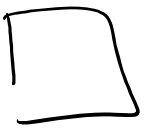
Let (v^*, w^*) be minimizing edge

- $X = X + w^*$
- $A[w^*] = A[v^*] + \ell_{v^* w^*}$ (already computed, since $v^* \in X$)
- $B[w^*] = B[v^*] + (v^*, w^*)$

Q: What is run time using adjacency list graph input?

```
for v ∈ V:
    if X[v] == 1:
        for w ∈ A_G[v]:
            if X[w] == 0:
                Calculate DGC
```

S. KIMMEL

- While Loop: n times
-  runtime: $O(m)$ times, since checks at most m edges

Total = $O(nm)$

* Use data structure to get speed up! *

Q: Which data structure is best for this algorithm

A) Stack B) Queue C) Heap/Priority Queue D) Binary Search Tree

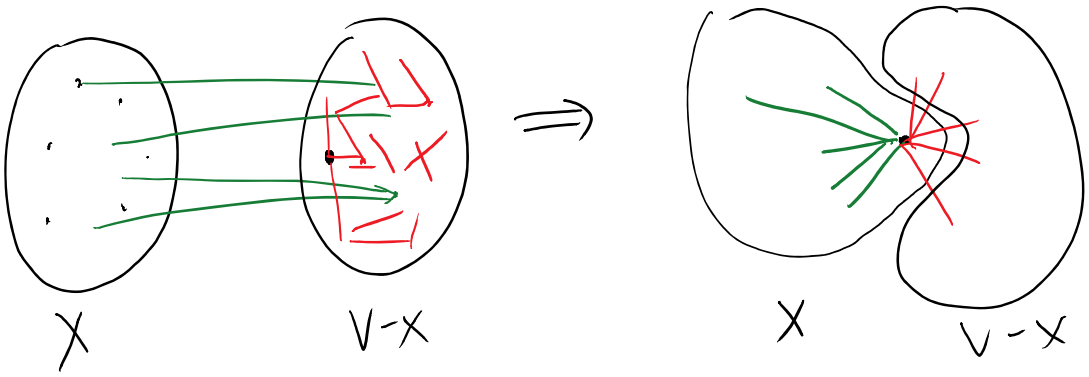
Repeated finding minimum, like heap sort

Operation	Time for heap with n items
Extract element with minimum key	O(log n)
Insert element	O(log n)
Learn minimum key	O(1)
Delete an element (if know position in heap)	O(log n)

S. KIMMEL

Q: What should be store in heap, and what should keys be? What is run time?

A: Store edges from X to $V-X$ in heap, with key given by $A[u] + \ell_{u,v}$ for edge (u,v)



- n times outer loop
- can have to do $O(n)$ insertions and deletions each round (for each edge connected from X to new vertex and each edge from new vertex to $V-X$)

$$n \quad n \quad \log m = O(n^2 \log m)$$

↑ rounds; ↑ insertion/deletion; ↑ time to insert delete

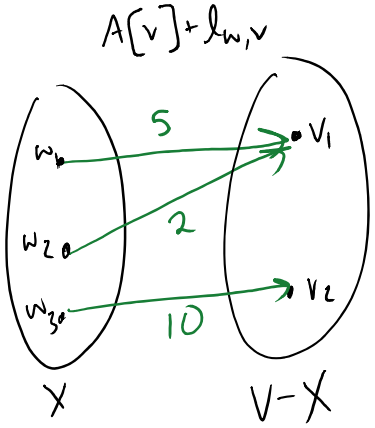
Hard to do deletions efficiently!

Better than $O(nm)$, but we can do better!

Instead

- Store vertices $v \in V-X$ in heap
- Key of v is

$$v.key = \min_{\substack{w \in X \\ (w,v) \in E}} \{A[w] + \ell_{w,v}\}$$

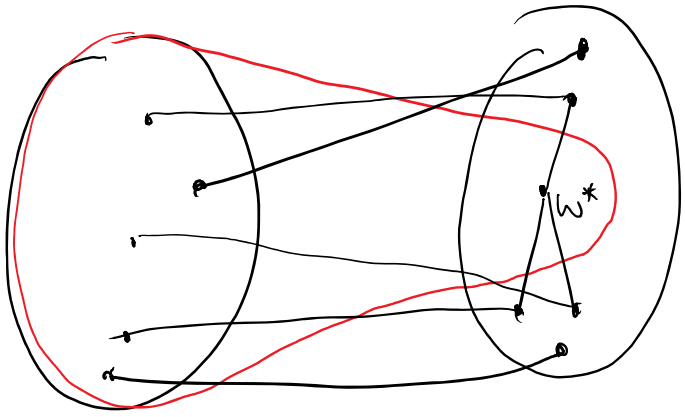


- For each element v,

$$v.p = \arg\min_{w \in X} \{A[w] + \ell_{wv}\}$$

- If v is not directly connected to X, $k(v) = \infty$, $v.p = \emptyset$

Why better?



$\Leftarrow$ Only need to check vertices in $A_G[w^*]$

X before w^* added

X after w^* added

S. KIMMEL

Dijkstra Heap

$X[v] = 0$; $A[v] = \infty$; $B[v] = \emptyset$; $\forall\ v \in V$

$v.key = \infty$ for all $v \in V$

$v.p = \emptyset$ for all $v \in V$

$s.key = 0$.

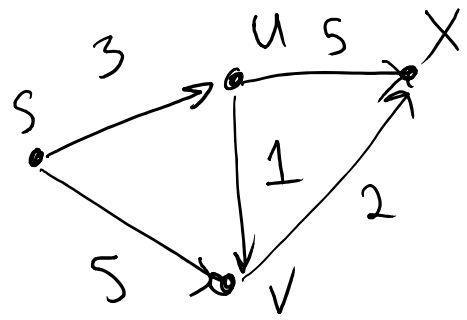
Heapify all $v \in V$

while (Heap is not empty)

- Let w = vertex with min key
- Remove w; $X[w] = 1$; $A[w] = w.key$;
- $B[w] = B[w.p] + (w.p, w)$;
- for $u \in A_G[w]$ && u not explored
  - Check if need to update $u.key$
  - If yes, remove and reinsert

S. KIMMEL

ex:



H

	Round 1	Round 2	Round 3
s	0 | ϕ	X	
u	∞ | ϕ	3 | s	X
v	∞ | ϕ	5 | s	4 | u → X
x	∞ | ϕ	∞ | ϕ	8 | u → 6 | v