Algorithms: Design and Analysis, Part II

Minimum Spanning Trees

Fast Implementation of Prim's Algorithm

Running Time of Prim's Algorithm

- initialize $X = \{s\}$ [$s \in V$ chosen arbitrarily]
- $T = \emptyset$ [invariant: X = vertices spanned by tree-so-far T]
- while $X \neq V$:
  - let $e = (u,v)$ be the cheapest edge of G with $u \in X$, $v \notin X$
  - add e to T, add v to X

Running time of straightforward implementation:

- $O(n)$ iterations [where n = # of vertices]
- $O(m)$ time per iteration [where m = # of edges]

$\Rightarrow O(mn)$ time

BUT CAN WE DO BETTER?

Speed-Up Via Heaps

Recall from Part I: raison d'être of a heap is to speed up repeated minimum computations
↳ seems useful for Prim's algorithm!

Specifically: a heap supports Insert, Extract-Min, and Delete in $O(\log n)$ time. [where n = # of objects in the heap]

Natural idea: use heap to store edges, with keys = edge costs.

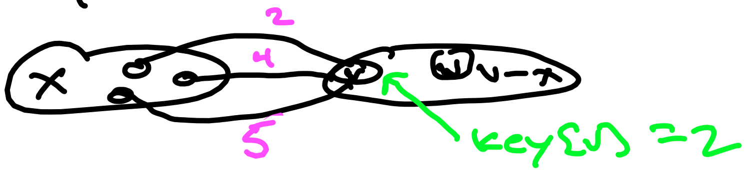
Exercise: leads to an $O(m \log n)$ implementation of Prim's algorithm.

Prim's Algorithm with Heaps

[compare to fast implementation of Dijkstra's algorithm]

Invariant #1: elements in heap = vertices of $V-X$

Invariant #2: for $v \in V-X$, key[v] = cheapest edge (u,v) with $u \in X$.



[or $+\infty$ if no such edges exist]

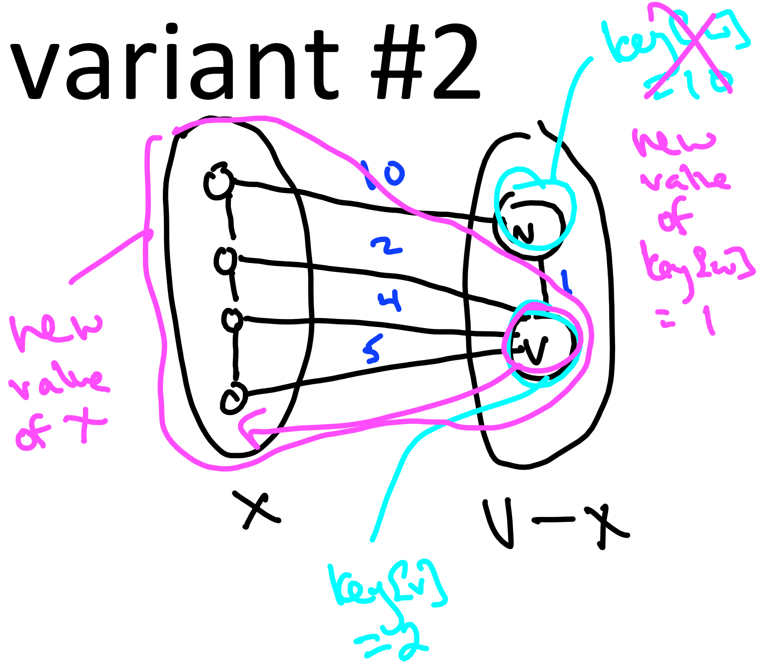
Check: can initialize heap with $O(\underbrace{m}_{\text{to compute keys}} + \underbrace{n \log n}_{n-1 \text{ inserts}}) = O(m \log n)$ preprocessing.

$m \geq n-1$ since G connected

Note: given invariants, Extract-Min yields next vertex $v \notin X$ and edge (u,v) crossing $(X, V-X)$ to add to X and T, respectively.

Quiz: Issue with Invariant #2

Question: what is: (i) current value of key[v] (ii) current value of key[w] (iii) value of key[w] after one more iteration of Prim's algorithm?



(A) 11, 10, 11

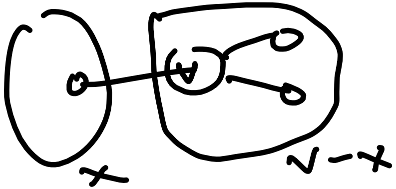
(B) 2, 10, 10

(C) 2, 10, 2

(D) 2, 10, 1

Maintaining Invariant #2

Issue: might need to recompute some keys to maintain Invariant #2 after each Extract-Min.



Pseudocode: when v added to X:
- for each edge $(v,w) \in E$:
  - if $w \in V - X$ → the only vertices whose key might have dropped
    - Delete w from heap
    - recompute $key[w] := \min\{key[w], c_{vw}\}$
    - re-Insert w into heap

update key if needed

Subtle point/exercise: think through book-keeping needed to pull this off

Running Time with Heaps

- dominated by time required for heap operations
- (n-1) Inserts during preprocessing
- (n-1) Extract-mins (one per iteration of while loop)
- each edge (v,w) triggers one Delete/Insert combo
  [when its first endpoint gets sucked into X]

$\Rightarrow$ O(m) heap operations [recall $m \geq n-1$ since G connected]

$\Rightarrow$ O(m log n) time [as fast as sorting!]