Week 1 Greedy algorithms 10/4/2018 1

Distributed shortest Path

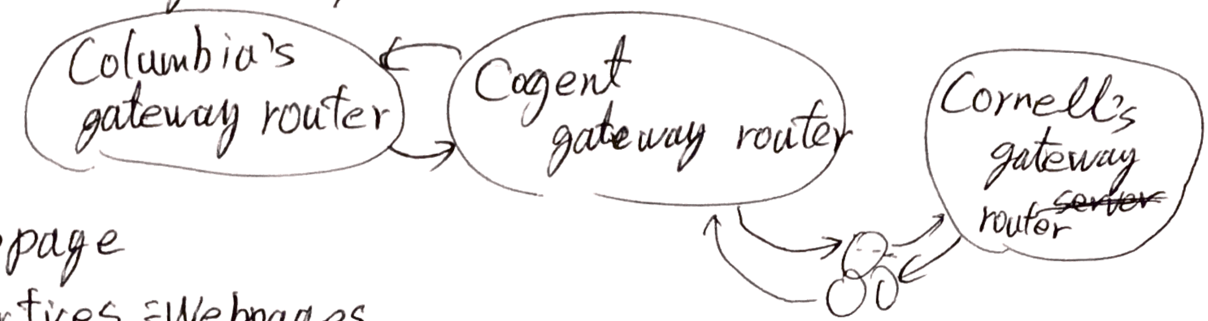
⇒ Why it is important?

⇒ Internet traffic.

Internet is a graph

- Vertices : end hosts + router
- Directed edges : physical or wireless connections

e.g.



- Webpage
  - Vertices : Webpages
  - Directed edges : Hyperlinks
- Social networks
  - vertices : People
  - Edges : Friend / Follow relationship.

How to get the shortest path?

⇒ Dijkstra?

⇒ Problem : In order to apply Dijkstra, Columbia gateway router need to know entire Internet graph.

⇒ Need a shortest-path algorithm that uses only local computation.

⇒ Bellman-Ford algorithm

Sequence Alignment

⇒ Operation to find how "similar" the two strings are.
DNA
(Used in computational genomics)
⇒ Eg. ① Extrapdate function of genomie substring.
② Check the proximity in evolutionary tree.

What does "similar" mean?

- A G G G C T
- A G G - C A

space mismatch

⇒ Can be nicely aligned.

⇒ Get the cost according to the cost and penalties of space and mismatches.

⇒ Needs the algorithm to compute fast enough to be useful. Brute-force search dose not work

⇒ Dynamic programming.

Algorithm ~~Dsg~~ Design Paradigms

⇒ No silver bullet.

Some paradigms

- Divide and conquer
- Randamized algorithms
- Greedy algorithms
- Dynamic programming

Greedy Algorithm

Def : Iteratively make "myopic" decisions, hope everything works out at the end;

Example: Dijkstra's shortest path

⇒ Processed each destination once, irrevocably. (No revisit)

Pros Cons Contrast with Divide & Conquer

1. easy to propose multiple greedy algorithm for many probs.
2. easy running time analysis. (Contrast with master method)
3. Hard to establish correctness.

Danger : Most greedy algorithms are not correct all the time. (~~unless~~ (Reguardless what intuitions says)

⇒ e.g. Dijkstra don't works when it has a negative number for the distances.

How I can prove it is correction ?

Method 1 : Induction

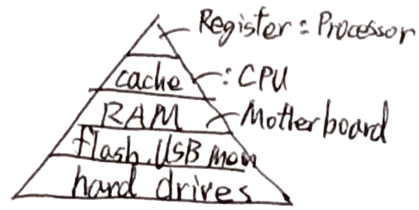
e.g. : Correctness proof for Dijkstra algorithm.

Method 2 : "exchange argument"

method 3 : Whatever it works.

Greedy application : Caching Problem

- Small fast memory (the ~~cha~~ cache)
- Big slow memory (~~Register~~, RAM)
- In the process of "page requests" ~~have to~~ insert data in caches.
- In a "fault" (cache miss), need to evict something from cache to make room.



Example

Cache | ~~a~~ e | ~~b~~ f | c | d |

Request sequence : c → d → e → f → a → oh, no! → b → Fuck!

- c: OK
- d: OK
- e: not in cache, evict 'a' and insert 'e'.
- f: Evict 'b' and insert 'f'.

⇒ 4 page faults
- 2 were inevitable (e, f)
- 2 consequences of poor eviction choices. (Should have evicted c & d instead of a & b)

⇒ Greedy algorithm is solution. (Furthest-in-future algo.)
⇒ Become a benchmark of algorithm.

Scheduling Problem

Setup: - One shared resource (e.g. a processor)
- Many "Jobs" to do (e.g. processes)

Q : In what order should we sequence the jobs?

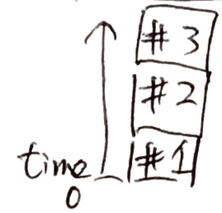
Assume: each job j has ① weight w_j ("Priority") ② length l_j

Def: The completion time C_j of job j = sum of job lengths up to and including j.

Example: 3 job, $l_1=1$, $l_2=2$, $l_3=3$

$C_1=1$ $C_2=3$ $C_3=6$

Schedule



Goal: Minimize the weighted sum of completion times: $\min \sum_{j=1}^{n} w_j c_j$

e.g. if $w_1=3$, $W_2=2$, $W_1=1$ total c time is

$3 \cdot 1 + 2 \cdot 3 + 1 \cdot 6 = 15$

Intuition

Purpose: Want to minimize $\sum_{j=1}^{n} w_j \cdot c_j$

Goal: Devise correct greedy algorithm

⇒ How to figure out the algorithm?

⇒ First, think about the special case.

If all the length (C_j) are same, prioritize the bigger W.

If all the weight (w_j) are same, prioritize the smaller C.

How to prioritize the task?

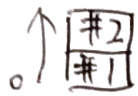
- Larger the weight ⇒ High prior.
  Smaller the length ⇒ High prior.

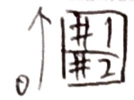
two way of getting cost.

① ~~w - e~~ $w - \ell$ (diff)

② $\frac{w}{\ell}$ (ratio) only one can always work.

e.g. $\ell_1 = 5$ $\ell_2 = 2$
$w_1 = 3$ $w_2 = 1$

ratio: $\frac{5}{3}$ > $\frac{1}{2}$  $5 \times 3 + 7 \times 1 = 22$ ⇒ Get smaller. Gave optimal

diff = -2 < -1  $2 \times 1 + 3 \cdot 7 = 23$ ⇒ Not always ~~not~~ end in optimal.

Prim's minimum spanning Tree algorithm (MST)

What? ⇒ Algorithm to connect a bunch of points together as cheeply as possible.

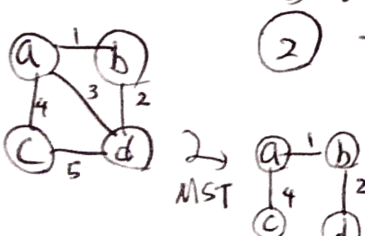
Application ⇒ clustering, networking

Input : Undirected graph $G = (V, E)$
- assume adjacency list representation.
- Ok if edges costs are negative.

Output : Minimum cost tree $T \subseteq E$ that spans all vertices.
↳ ie., sum of edge costs.

① T has no cycles  disallowed

② the subgraph (V, T) is connected
(i.e., contains path between each pair of vertices)



cost = 7

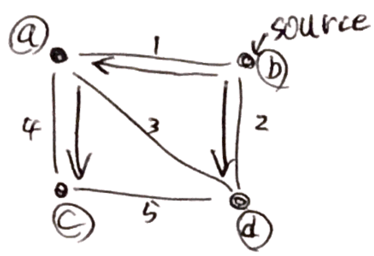
Assumption #1: Input graph G is connected

- Else no spanning trees.
- Easy to check in preprocessing (DFS, BFS)

Assumption #2: Edges costs are distinct

- Prim + Kruskal remain correct with ties.

Algorithm



1. Pick source node randomly. (b)
2. Choose the cheaper edge (Greedy) and expand to a. Now tree is a-b
3. From a-b, 3 options to expand which are (c, 4), (d, 3), (d, 2).
4. Choose the cheapest one which is (d, 2). Now the tree is a-b-d.
5. From a-b-d, 2 options to expand (c, 4) (c, 5).
6. Choose the cheaper one which is (c, 4)
7. all nodes connected! Done!

Pseudo code

- initialize $X = [s]$ (Explored; ~~start node~~) $[s \in V$ chosen arbitrarily$]$
- T (tree) = None [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

i.e. increase # of spanned vertices in cheapest way possible.

Implementation of MST ~~/~~ Prim's algorithm. ⑧

⇒ Running time of Prim's

linear $\begin{cases} \text{- initialize } X = [s] \\ - T = \emptyset \end{cases}$

loop for $n-1$ times ⇒ $O(n)$

- While $X \neq V$:
  - let $e = (u,w)$ the cheapest } Use "heap" to get min in $O(\log m)$.
  - Add e to T, add v to X

How to implement the heap?

- Natural idea: Use heap to store edges, with keys = edge cost

⇒ Better idea?

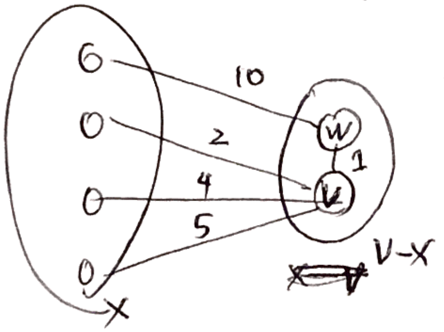
⇒ Store the nodes!

How to store the nodes in heap?

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

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

Check: Initialize the heap with $O(m \log n)$ preprocessing.

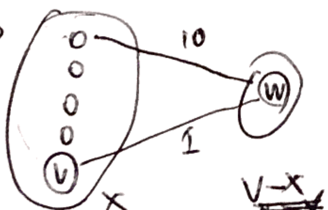


Current key$[v]$ = 2 (cheapest in 2, 4, 5)

current key$[w]$ = 10 (Only edge)

Next iteration key$[w]$ = 1 (Cheapest in 1, 10)

Next iteration



⇒ Have to recompute the key every iteration.

How to update the key value?

when v added to X:

- for each edge $(v,w) \in E$:
  - if $w \in V-X$ → the only vertices whose key might have changed
    - Delete w from heap
    - Recompute $key[w] :=$ $\min[key[w], Cost_{vw}]$
    - Re-Insert w into heap

update key if needed.

$\Rightarrow O(m \log n)$