Greedy Algorithm Cont.

Frank

September 21, 2021

1 Dijkstra's Algorithm KT 4.4/DPV 5.4

1.1 Input

- A Weighted, directed graph $G = (V, E)$
- "source" node $s \in V$
- $wt : E \to \mathbb{R}$ s.t. $wt(e) \geq 0$
- wt of path $P = v_1, v_2, \ldots, v_k$, $wt(P) = \sum_{1 \leq i < k} wt(v_i, v_{i+1})$

1.2 Output

$$\forall v \in V, \text{ compute } \delta(v) = \begin{cases} \text{min } wt \text{ of } s \to v \text{ path}, & \text{if such path exists} \\ \infty, & \text{if no such path exists} \end{cases}$$

1.3 Algorithm

- Maintains
  - a set $R \in V$: "explored" region of graph
  - a quantity attached to every node $d(v)$: min wt of $s \to v$ path in which all nodes $\neq v$ are in R. Define underlined as "R-path"

And maintains it in a way such that the following are true

1. $\forall v \in V$, $d(v) =$ min wt of an R-path from s to $v \implies d(v) \geq \delta(v)$
2. $\forall v \in R$, $d(v) = \delta(v)$

- Dijkstra's algorithm is essentially a greedy algorithm that starts with a trivial R, and greedily expands R one node at a time until it has explored the entire graph and has found minimum weight path of every node starting from s

Algorithm:

```
1: procedure DIJKSTRA
2:     R ← ∅
3:     d(s) ← 0
4:     p(s) ← null                                          ▷ predecessor
5:     for (every node v ≠ s) do
6:         d(v) ← ∞
7:         p(v) ← null
8:     while (R ≠ V) do
9:         u ← node in V − R s.t. d(u) is minimum      ▷ look for min in unexplored paths
10:        R ← R ∪ {u}                                        ▷ add to explored paths
11:        for (each v s.t. (u, v) ∈ E) do                ▷ look at its out neighbours
12:            if (d(v) > d(u) + wt(u, v)) then
13:                d(v) ← d(u) + wt(u, v)
14:                p(v) ← u
```

Note that at line 10, we expand R, and decrease the size of $V - R$. This means new R-paths may be generated for nodes v adjacent to u, and $d(v)$ may be updated for these nodes by comparing the new path with the old path (line 12).

Note that for line 11, we can look at out going edges to nodes in R as well as $V - R$ (so all of V), but it is actually a useless step, and we only need to look at out going edges to nodes in $V - R$

At the end, use the $p(v)$ to back trace every node to s to find the minimal weighted path

1.4 Running Time

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

1.4.1 Naive implementation

- Without use of heap gives $O(n^2)$, since while takes $O(n)$, line 9 takes $O(n)$, and line 11 takes at most $O(n)$

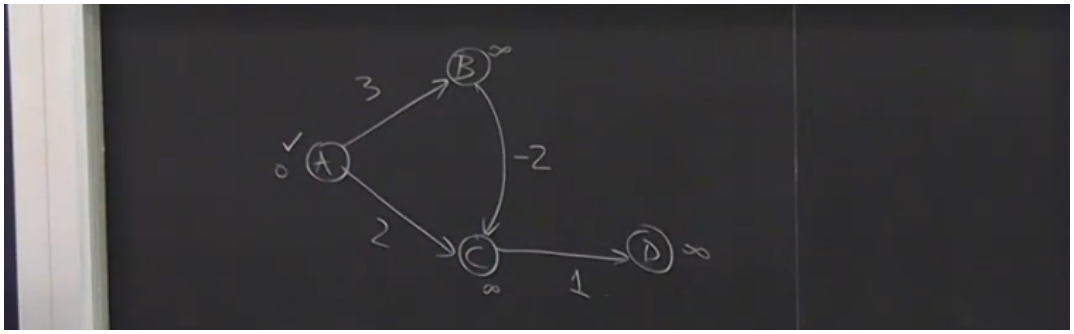
1.4.2 Sophisticated implementation

- Keep nodes in $V - R$ in a heap
  - EXTRACTMIN $O(\log n)$ n times (line 9)
  - CHANGEKEY $O(\log n)$ m times (line 13)
- Initialize heap with $O(n)$
- Then gives $O(n \log n + m \log n) \stackrel{*}{=} O(m \log n)$ with the assumption that there is a path from source to every other node

1.4.3 Naive vs Sophisticated implementation

- In the worst case, $m \sim O(n^2)$, so our sophisticated implementation is actually worse in the worst case.
- But most graphs we deal with are "sparce", and m is no way near its worst case, so our sophisticated implementation is better in most cases.

1.5 Algorithm failure due to negative weights



Trace above to find that we actually broke a greedy algorithm rule, we will put c in R, but then find out there's a better way from b to c, but then d will never get corrected, since c is already in R. Thus, the algorithm fails, we will not have the correct $\delta(v)$ values

1.6 Proof of Correctness

1.6.1 Claim 1

$\forall v \in V$, $d(v)$ is non-increasing through the iterations $[i \leq j \implies d_i(v) \geq d_j(v)]$

- Obvious from line 12-13 of algorithm

1.6.2 Claim 2

If u is added to R in iteration i, then $d_i(u) = d_{i-1}(u)$

- In other words, the distance value of u never changes once it is added to R. This is good because when u is added to R, its neighbours are updated, so if then u is updated again sometime in the future, then its neighbours may have the wrong d value.

  Note that the d value of u never changes from the iteration right before it was added to R to the iteration when it was added to R $[d_i(u) = d_{i-1}(u)]$

1.6.3 Claim 3

At the end of iteration i, if $d_i(v) = k \neq \infty$, then $\exists$ an R-path $s \to v$ of weight k

- Should be obvious, but even if we change $d(v)$, it is because there is a new (better) R-path to replace the old R-path

1.6.4 Claim 4

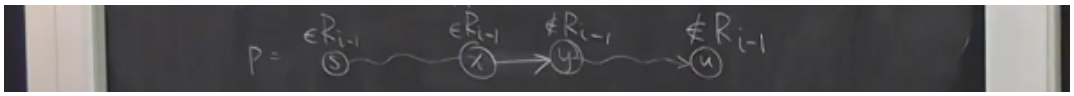
If $d(u) = \infty$ when u is added to R, then $\nexists$ $s \to u$ path in the graph

- Proof

  Suppose for contradiction, that node u is added to R in iteration i and $d_i(u) = \infty$, but $\exists$ $s \to u$ path p.

  Without loss of generality (WLOG), assume i is the earliest iteration that above happens. Note $i > 1$, since the first node put into R is the source node s, and $d(s) = 0$

Consider iteration $i-1$, we have a path $s \to u$, note that $s \in R_{i-1}$ since s is put into R in $i = 1$ and $i > 1$. However, $u \notin R_{i-1}$ because u is put into R at iteration i. This means at some point we hit an edge (x, y) s.t. $x \in R_{i-1}$ and $y \notin R_{i-1}$



Let j be the iteration when x was added to $R \implies j \leq i-1 \implies d_j(x) \neq \infty$ [by def of i] $\implies d_j(y) \neq \infty$, because if it was, it will be changed by algorithm $\implies d_{i-1}(y) \neq \infty$ by claim 1.

So now we have a node $y \neq \infty \notin R$ but yet added a node with d value of infinity, which contradicts our algorithm.

$\therefore$ claim 4 holds

1.6.5 Claim 5

If u is added to R in iteration i, then $d_i(u) = \delta(u)$ [$d_i(u)$ has correct value]

- Proof: Complete induction on i

When $u = s$, the claim is trivially true, since $d(s) = 0 = \delta(s)$

If $d_i(u) = \infty$ then by claim 4, we're done, since there is no path from s to u

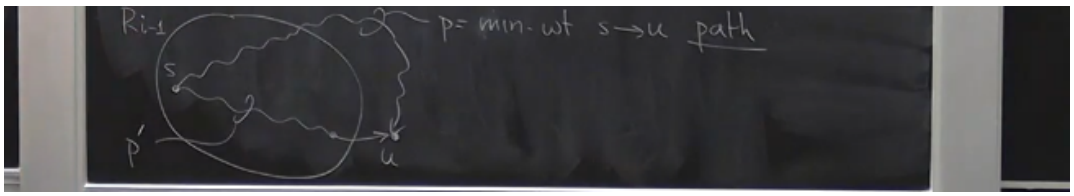
So suppose $d_i(u) = k \neq \infty \implies$ By claim 3, $\exists$ R_{i-1}-path p' from $s \to u$ of weight equal to k

$\implies d_i(u) \geq \delta(u)$. And it is enough to show that $d_i(u) \leq \delta(u)$

Consider set R_{i-1}, R just before the iteration where we add u to it.

Consider an R_{i-1}-path p' that goes from s to u with weight k

Consider a min-weight path p from s to u, then weight of this path equals to $\delta(u)$. Since u is not in R_{i-1}, at some point, the path should go outside R_{i-1} and create a cut edge (x, y), where y is the first node in p that is not in R_{i-1} and x is the predecessor of y on $p \implies x \in R_{i-1}$.



Let j be the iteration when x was added to $R \implies j \leq i-1$

$d_i(u) = d_{i-1}(u)$ by claim 2

$\leq d_{i-1}(y)$ by algorithm, since u was added on iteration i, meaning it has the smallest d value amongst all the choices

$\leq d_j(y)$ by claim 1

$\leq d_j(x) + wt(x, y)$ by algorithm

$= \delta(x) + wt(x, y)$ by IH

$= wt(\text{prefix of } p \text{ up to } x) + wt(x, y)$

$\leq wt(p)$ because edge weights are non-negative

$= \delta(u)$

$\implies d_i(u) \leq \delta(u)$

$d_i(u) \geq \delta(u)$ and $d_i(u) \leq \delta(u) \implies d_i(u) = \delta(u)$ as wanted

1.6.6 Thm

When the algorithm terminates, $d(u) = \delta(u)$

- Proof: By claim 5, when u is added to the set R, $d(u) = \delta(u)$, by claim 1, we cannot make $d(u)$ any bigger, by claim 2, we cannot make $d(u)$ any smaller. So it will stay optimal until the end. $\square$

2 Fractional Knapsack Problem

2.1 Problem

- We have a store that has n items $1, 2, \ldots, n$
- Item i has physical weight $w_i > 0$ and value $v_i > 0$
- A thief breaks into the store and has a knapsack of capacity W
- The theif will steal a fraction of each item in the store, define that fraction to be x_i such that $0 \leq x_i \leq 1$: fraction of item i to be stolen
- Then the fraction of the item i will have weight $w_i \cdot x_i$ and value $v_i \cdot x_i$
- Define Knapsack $S = (x_1, \ldots, x_n)$ is decision on what fraction of each item will be stolen. We have requirements:

  a) $0 \leq x_i \leq 1$

  b) $\sum_t w_t x_t \leq W$

- Define value of the knapsack S to be $V(S) = \sum_t v_t x_t$

2.2 Input/Output

- Given w_i, v_i s.t. $1 \leq i \leq n$ and W
- Return a knapsack of max value, $max(V(S))$

2.3 0-1 (discrete) knapsack

- Replace

  a') $x_i = 0$ or $x_i = 1$. (all or nothing)

- This is a NP-complete problem

2.4 Algorithm

1: **procedure** GREEDY
2: Sort items in non-increasing $\frac{value}{wt}$ ratio
3: Take entire items in this order until next item k would exceed W
4: Take fraction of item k that still fits W
5: Take 0 of all remaining items

- Note line 2, we are trying to maximize value and minimize weight, so take the highest value to weight ratio.

  Value being higher increases $\frac{value}{wt}$

  Weight being lower increases $\frac{value}{wt}$

- Sample output should look like

$$G = (1, 1, \ldots, 1, x_k, 0, 0, \ldots, 0)$$

To be continued next lecture...