CS 341 – Algorithms

Lecture 10 – Single Source Shortest Paths

18 June 2021

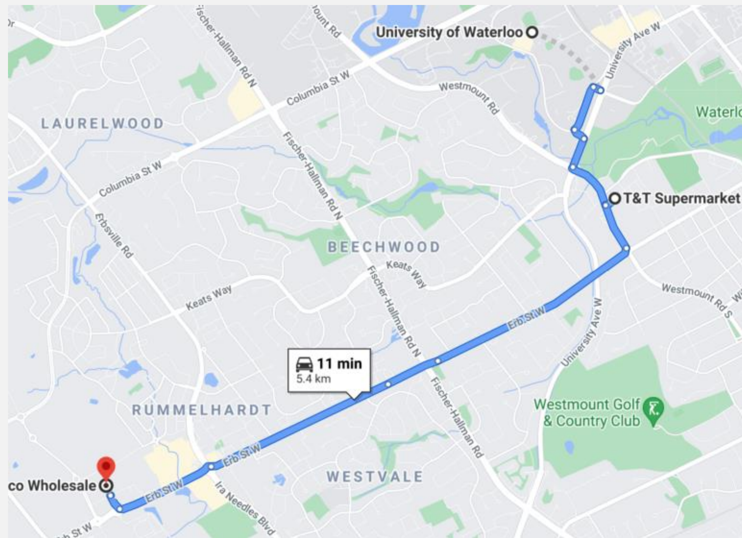
Today’s Plan

1. Dijkstra’s algorithm as simulating BFS
2. Dijkstra’s algorithm as a greedy algorithm

Single Source Shortest Paths

Input: A directed graph $G = (V, E)$, a **non-negative** length l_e for each edge $e \in E$, and two vertices $s, t \in V$.

Output: A shortest path from s to t, where the length of a path is equal to the sum of the length of its edges.



(SSSP)

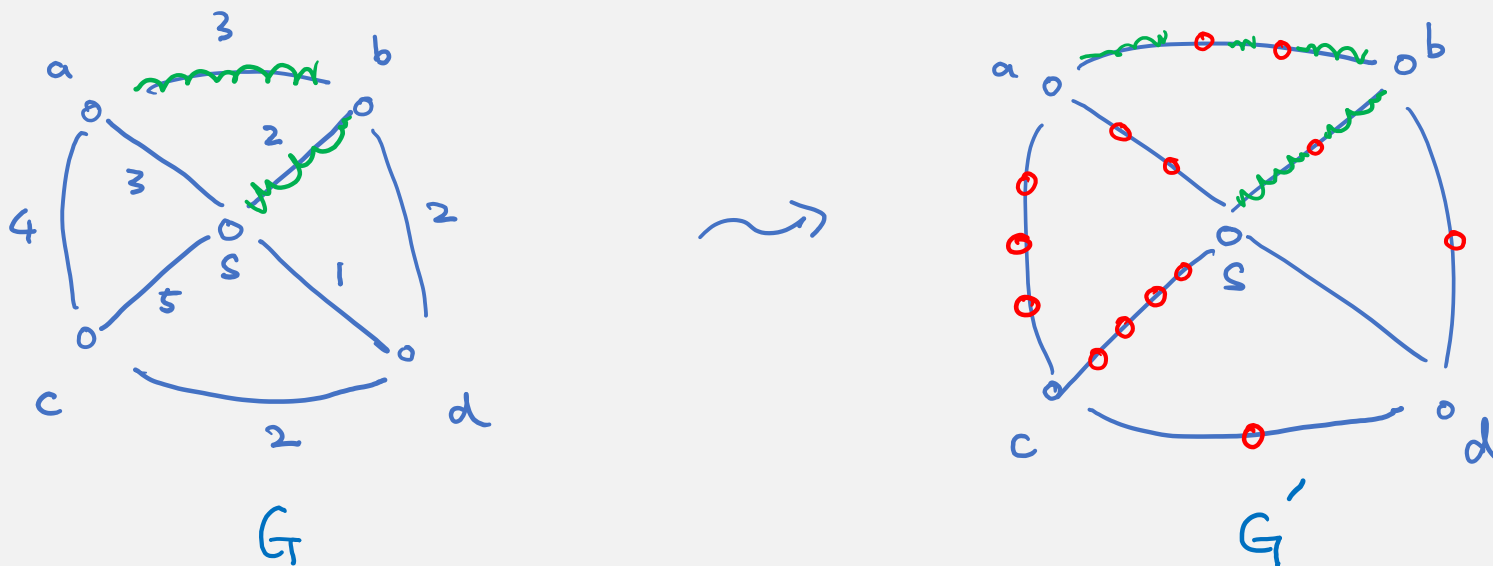
Input: A directed graph $G = (V, E)$, a **non-negative** length l_e for each edge $e \in E$, and a vertex $s \in V$.

Output: A shortest path from s to v, for every vertex $v \in V$. ← $\Omega(n^2)$?

Breadth First Search

In L05.pdf, we see that if every edge has length one, then the problem can be solved by BFS.

We can reduce the non-negative length problem to this special case.



Claim: $\exists$ a path of length k from s to v in G

$\Leftrightarrow$ $\exists$ a path of length k from s to v in G'

The Reduction

What is wrong with the reduction?

nothing wrong with the correctness

but it is not an efficient reduction

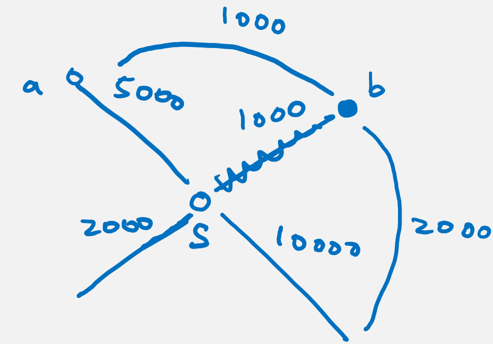
$$\# \text{ vertices in } G' = \sum_{e \in E(G)} (l_e - 1) + n$$

$l_e = 1000000$, $l_e = n^2$, $l_e = 2^n$

idea: just keep G' and BFS in mind,

but we want to efficiently simulate the process

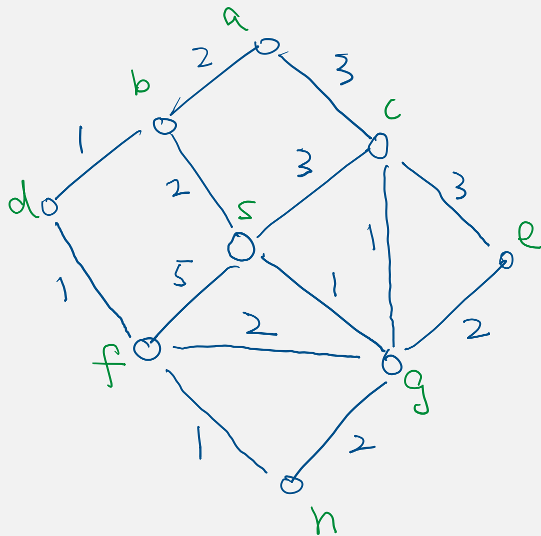
Physical Process

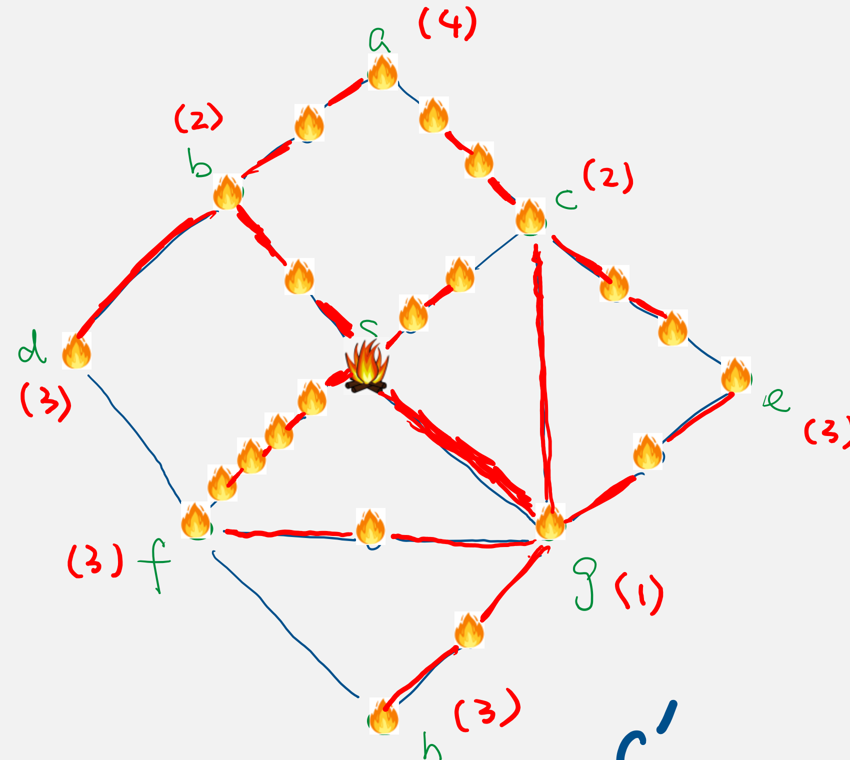


We can think of the process of BFS in G' as follows.

- We start a fire at vertex s at time 0.
- It takes one unit of time to burn an edge e in G'.

Claim. The shortest path distance from s to t is just the first time when vertex t is burnt.

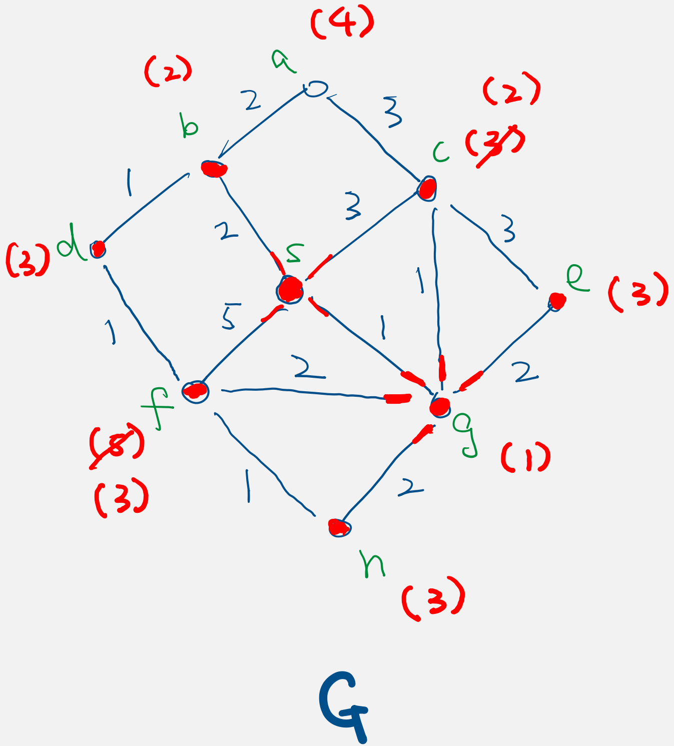


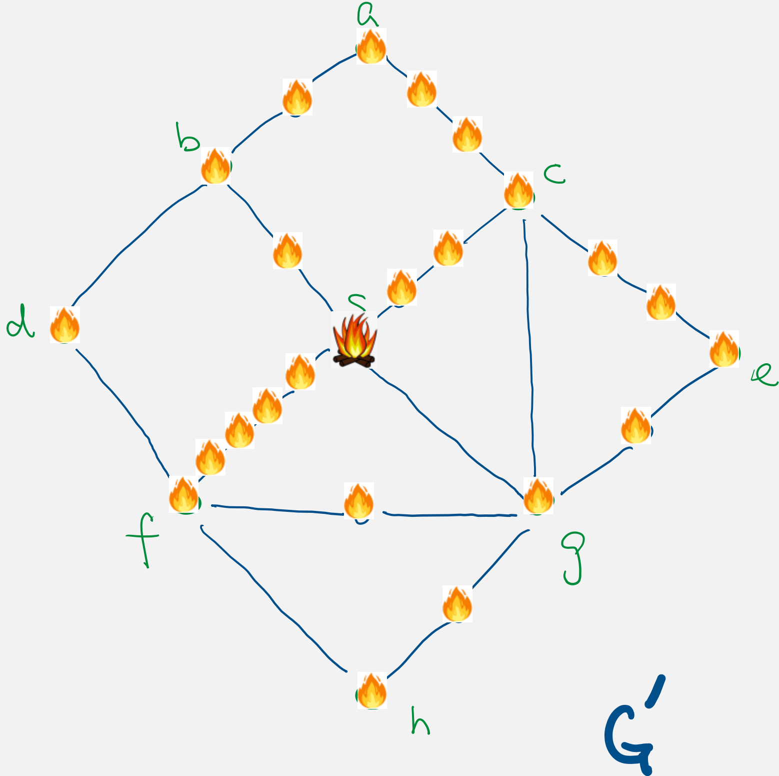


Efficient Simulation

The idea is that we just need to be able to keep finding out what is the next vertex to be burnt and when,

by keeping track of an **upper bound** on the time a vertex to be burnt.





Dijkstra's Algorithm as BFS Simulation

```
dist(v) = ∞  for every v ∈ V          // the initial upper bound on the time v is burned

dist(s) = 0                           // start a fire at vertex s at time 0

Q = make-priority-queue(V)            // all vertices are put in the priority queue.
                                      //   using dist(v) as the key value (priority value) of v

While Q is not empty do

    u = delete-min(Q)                 // dequeue the vertex with minimum dist() value
                                      //   i.e. find out the next vertex to be burned

    for each (out-)neighbor v of u    // it works for directed graphs as well

        if dist(u) + l_uv < dist(v)   // find a shorter way to get to v through u

            dist(v) = dist(u) + l_uv

            decrease-key(Q, v)

            parent(v) = u             // note that this may be updated multiple times
```

Analysis

Fibonacci heap

$O(m + n\log n)$

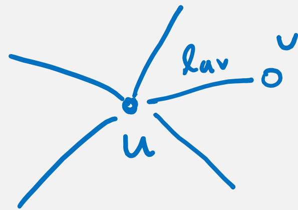
Correctness : Shortest path in G $\Leftrightarrow$ shortest path in G'

↑ unweighted . BFS

our algo simulated BFS in G'.

time complexity: each vertex is enqueued once (beginning) ← $O(n\log n)$ time

is dequeued once (loop)



check $dist(u) + l_{uv} < d(v)$

if so, decrease-key

priority-queue

240 by a min-heap

⇓

enqueue dequeue

decrease-key

$O(\log n)$ time

each iteration : $O(\deg(u) \cdot \log n)$

total time : $O(n\log n + \sum_v \deg(v) \cdot \log n) = O((n+m)\log n)$. □

Today’s Plan

1. Dijkstra’s algorithm as simulating BFS

2. Dijkstra’s algorithm as a greedy algorithm

Traditional Approach

Here we follow a more traditional approach to prove the correctness of Dijkstra's algorithm.

Besides being more formal, the proof technique is also useful in analyzing other problems, e.g. MST.

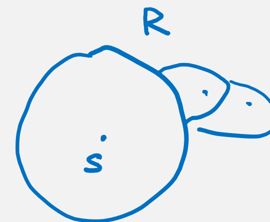
We think of Dijkstra's algorithm as a greedy algorithm.

The idea is to grow a subset $R \subseteq V$ so that $dist(v)$ for $v \in R$ are computed correctly.

Initially, $R = \{s\}$, and then at each iteration we add one more vertex to R.

Which vertex to be added in an iteration?

We will add the vertex which is closest to R to R, so in this sense the algorithm is greedy.



Dijkstra's Algorithm as Greedy Algorithm

$dist(v) = \infty \quad \forall v \in V. \qquad dist(s) = 0.$

$R = \phi.$

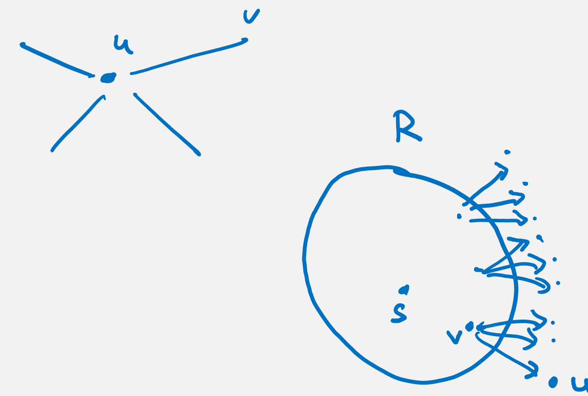
While $R \neq V$ do

pick the vertex $u \notin R$ with smallest $dist(u)$. $R \leftarrow R \cup \{u\}$.

for each edge $uv \in E$

if $dist(u) + l_{uv} < dist(v)$

$dist(v) = dist(u) + l_{uv}.$



Note that $u \notin R$ is a vertex with $dist(u) = \min_{v \in R, w \notin R} \{dist(v) + l_{vw}\}$.

Correctness by Induction

Invariant: For any $v \in R$, $dist(v)$ is the shortest path distance from s to v.

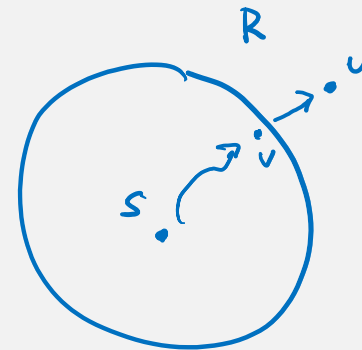
base case: $R = \{s\}$. $dist(s) = 0$ ✓

induction step: computed correctly for vertices in R.

add u to R, show computed correctly for vertices in $R + \{u\}$.

goal: shortest path from s to u $= dist(v) + l_{vu}$

① shortest path from s to u $\leq dist(v) + l_{vu}$



by I.H.

$\exists$ a path P_{sv} with $length(P_{sv}) = dist(v)$

$\Rightarrow$ $P_{sv} + vu$ is a path from s to u with length $dist(v) + l_{vu}$.

let v, u be ~~the~~ a minimizer

Note that $u \notin R$ is a vertex with $dist(u) = \min_{v \in R, w \notin R} \{dist(v) + l_{vw}\}$.

Correctness by Induction

Invariant: For any $v \in R$, $dist(v)$ is the shortest path distance from s to v.

goal:

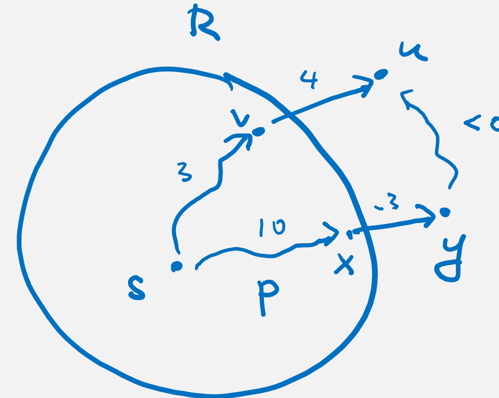
② shortest path from s to u $\geq dist(v) + l_{vu}$

Consider any path P from s to u

(notice x could be v. y could be u)

$$length(P) \geq dist(x) + l_{xy} + \underbrace{(\dots)}_{\geq 0}$$

$$\geq dist(v) + l_{vu}$$

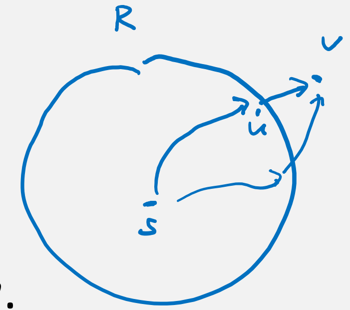


① + ② ⇒ $dist(v) + l_{vu}$ is the shortest path distance from s to u. □

v, u is a minimizer

Note that $u \notin R$ is a vertex with $dist(u) = \min_{v \in R, w \notin R} \{dist(v) + l_{vw}\}$.

Shortest Path Tree

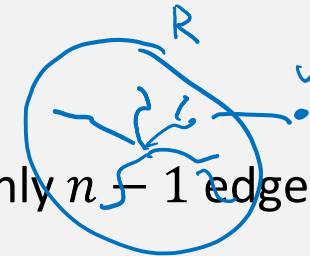


From the proof, when a vertex v is added to R, any vertex $u \in R$ that

minimizes $dist(u) + l_{uv}$ is the parent of v on a shortest path from s to v.

We keep track of these parent information, which can be used to trace back a shortest path from s.

As in BFS tree, these edges $(v, parent[v])$ form a tree.

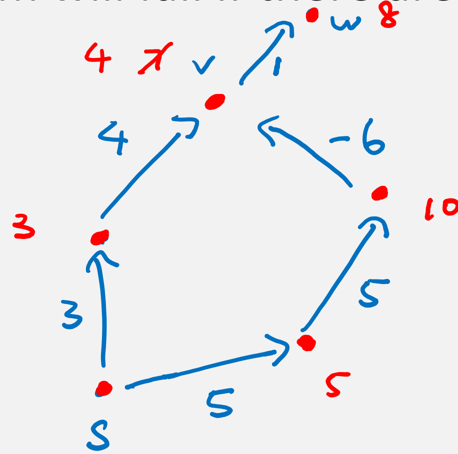


So, we have a succinct way to store all the shortest paths from s using only $n - 1$ edges.

The shortest path tree is a nice structure and is very useful algorithmically.

Negative Edge Length?

Question: How the algorithm will fail if there are some negative edge lengths?



We will come back to this more general setting after we introduce dynamic programming.

Concluding Remarks

We have seen a few greedy algorithms.

The exchange argument is useful because it allows us to compare two similar solutions.

This is the end of the third topic of the course, exactly finishing half of the lectures.

Next time, we will introduce the technique of dynamic programming.

The second half of the course will be more advanced and interesting.