Single-Source Shortest Paths

Lecture 18

April 1, 2025

Part I

Introduction to SSSP

Shortest Paths in the Wild

Common problem: “what’s the most efficient way to get from point A to point B?”

Shortest Paths in the Wild

Common problem: "what's the most efficient way to get from point A to point B?"

- What's the fastest route from Siebel to Lincoln hall?

Shortest Paths in the Wild

Common problem: "what's the most efficient way to get from point A to point B?"

- What's the fastest route from Siebel to Lincoln hall?
- How many network hops does a packet take to get from the 374 website server to your computer?

Shortest Paths in the Wild

Common problem: "what's the most efficient way to get from point A to point B?"

- What's the fastest route from Siebel to Lincoln hall?
- How many network hops does a packet take to get from the 374 website server to your computer?
- ...

Shortest Paths in the Wild

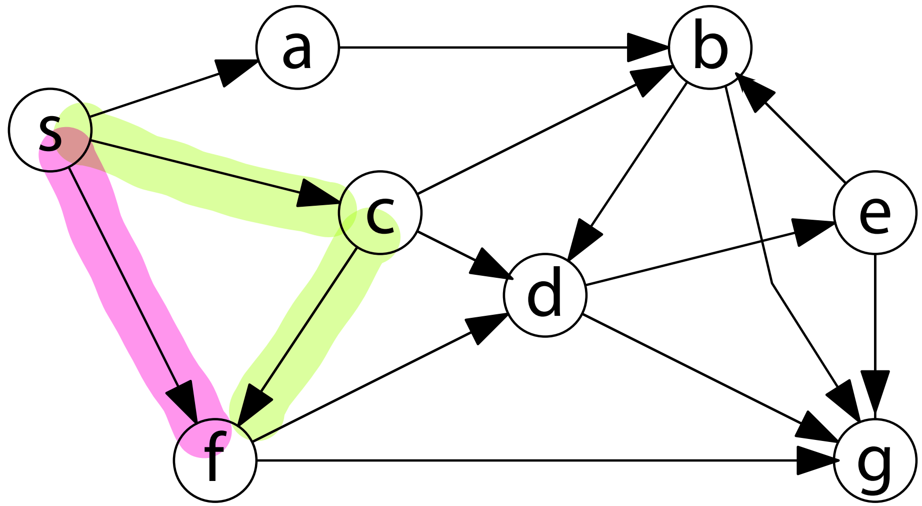
Common problem: "what's the most efficient way to get from point A to point B?"

- What's the fastest route from Siebel to Lincoln hall?
- How many network hops does a packet take to get from the 374 website server to your computer?
- ...

Goal for this week: define and solve these problems in graphs.

Formal Definitions

Given a graph $G = (V, E)$ and two nodes s, t the *distance* $\text{dist}(s, t)$ is the length of the shortest path from s to t in G.



$\text{dist}(s, s) = 0$

$\text{dist}(s, c) = 1$

$\text{dist}(s, f) = 1$

Formal Definitions

Given a graph $G = (V, E)$ and two nodes s, t the *distance* $\mathrm{dist}(s, t)$ is the length of the shortest path from s to t in G.

Problems of interest:

- Given s and t, find $\mathrm{dist}(s, t)$.
- Given s, find $\mathrm{dist}(s, t)$ for *all* t. SSSP
- Find $\mathrm{dist}(s, t)$ for *all pairs* of s and t. APSP

Formal Definitions

Given a graph $G = (V, E)$ and two nodes s, t the *distance* $\mathrm{dist}(s, t)$ is the length of the shortest path from s to t in G.

Problems of interest:

- Given s and t, find $\mathrm{dist}(s, t)$.
- Given s, find $\mathrm{dist}(s, t)$ for *all* t.
- Find $\mathrm{dist}(s, t)$ for *all pairs* of s and t.

Common variations:

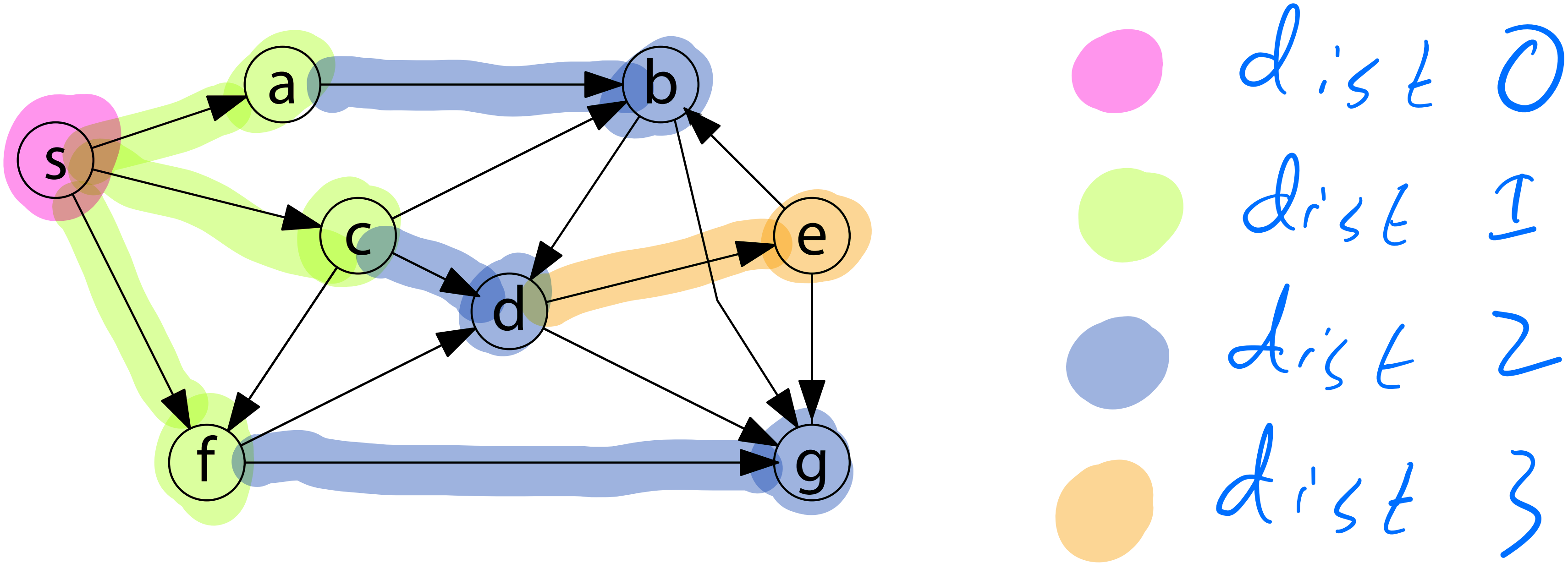
- Directed vs undirected graphs
- Weighted vs unweighted edges

Part II

Unweighted Graphs

Building Intuition

What vertex do we *definitely* know the distance to? (Don't overthink this!)



Intuitive Algorithm

```
IntuitionUSSSP(G, s):
    Label s with distance 0
    Set d = 0
    while there exists a vertex labeled with distance d:
        for all vertices u labeled with distance d:
            for all edges (u, v):
                if v has not been labeled:
                    Label v with distance d + 1
        d = d + 1
```

Intuitive Algorithm

```
IntuitionUSSSP(G, s):
    Label s with distance 0
    Set d = 0
    while there exists a vertex labeled with distance d:
        for all vertices u labeled with distance d:
            for all edges (u, v):
                if v has not been labeled:
                    Label v with distance d + 1
        d = d + 1
```

Claim: `IntuitionUSSSP` is really just `BFS` in disguise!

Intuitive Algorithm

```
IntuitionUSSSP(G, s):
    Label s with distance 0
    Set d = 0
    while there exists a vertex labeled with distance d:
        for all vertices u labeled with distance d:
            for all edges (u, v):
                if v has not been labeled:
                    Label v with distance d + 1
        d = d + 1
```

Claim: `IntuitionUSSSP` is really just BFS in disguise!

- BFS starts with all vertices at distance 0 in `ToExplore`. (just s)

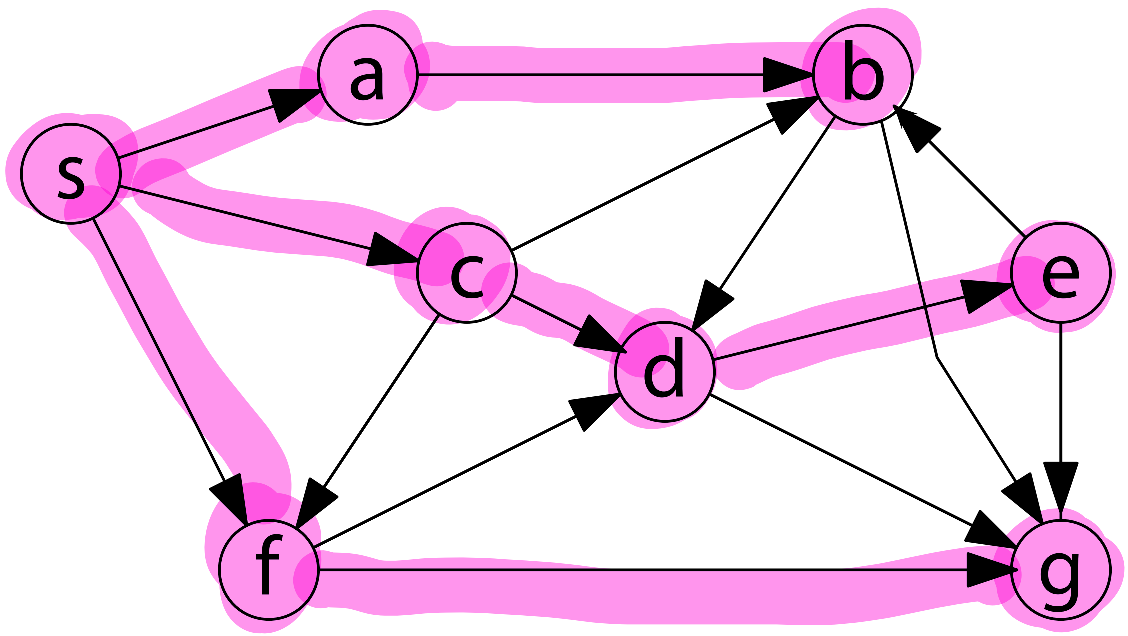
Intuitive Algorithm

```
IntuitionUSSSP(G, s):
    Label s with distance 0
    Set d = 0
    while there exists a vertex labeled with distance d:
        for all vertices u labeled with distance d:
            for all edges (u, v):
                if v has not been labeled:
                    Label v with distance d + 1
        d = d + 1
```

Claim: `IntuitionUSSSP` is really just `BFS` in disguise!

- `BFS` starts with all vertices at distance 0 in `ToExplore`. (just s)
- While processing vertices at distance d, `BFS` adds all vertices at distance $d + 1$ to the end of the `ToExplore` queue.

BFS Example



ToExplore = { s ¦ a, c, f ¦ b, d, g ¦ e ¦ }

dist 0 1 2 3

Unweighted Case Takeaways

If G is unweighted, we can use BFS to solve the SSSP problem in $O(V + E)$ time.

Exercise: modify the BFS pseudocode to output the shortest path distances, then to output the paths themselves.

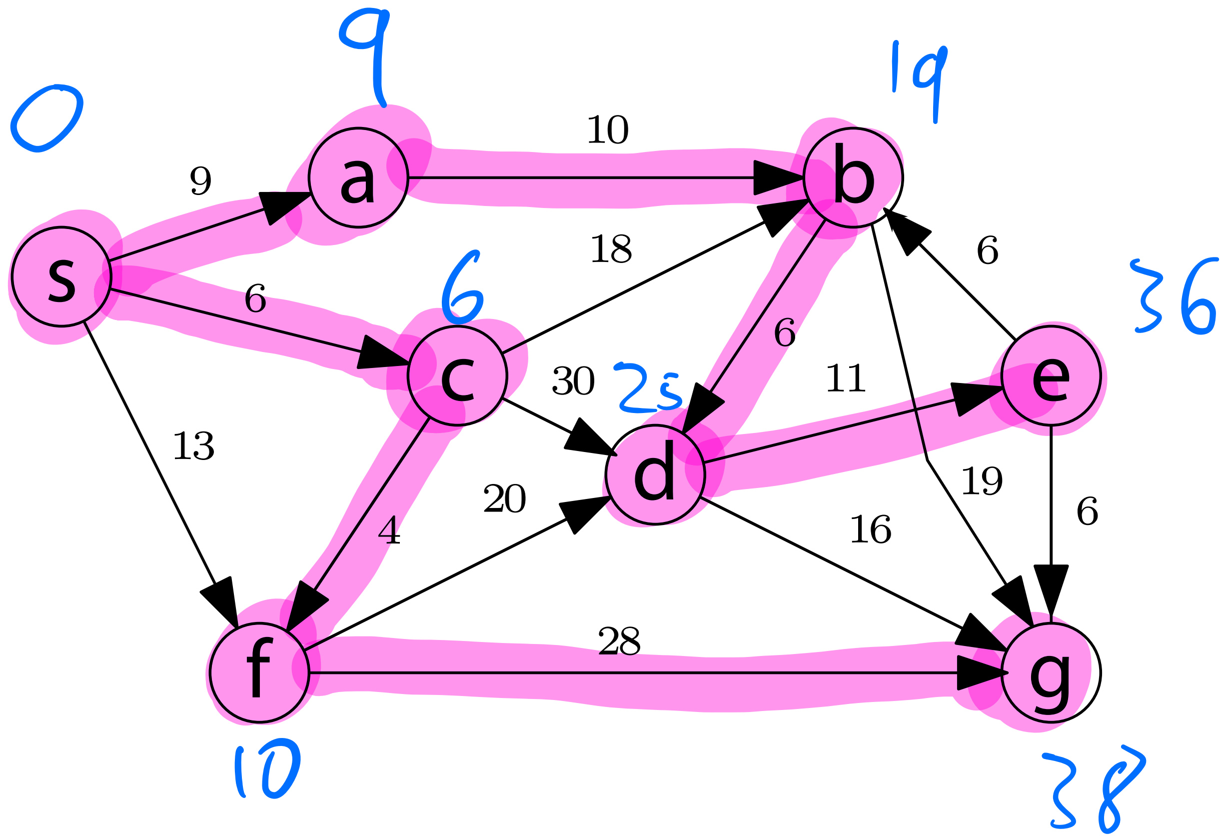
(Hint: start from the modification that outputs the BFS tree.)

Part III

Weighted Graphs

Building Intuition

What vertex do we *definitely* know the distance to? (Don't overthink this!)



Intuitive Algorithm

```
IntuitionWSSSP(G, s):
    Set s.guess = 0
    while there exists a vertex with a guess but no dist:
        Let u be such a vertex with the smallest guess
        Set u.dist = u.guess
        for each edge (u, v):
            if v has a guess:
                Set v.guess = min(v.guess, u.dist + w(u, v))
            else
                Set v.guess = u.dist + w(u, v)
```

Intuitive Algorithm

```
IntuitionWSSSP(G, s):
    Set s.guess = 0
    while there exists a vertex with a guess but no dist:
        Let u be such a vertex with the smallest guess
        Set u.dist = u.guess
        for each edge (u, v):
            if v has a guess:
                Set v.guess = min(v.guess, u.dist + w(u, v))
            else
                Set v.guess = u.dist + w(u, v)
```

Correctness?

Intuitive Algorithm

```
IntuitionWSSSP(G, s):
    Set s.guess = 0
    while there exists a vertex with a guess but no dist:
        Let u be such a vertex with the smallest guess
        Set u.dist = u.guess
        for each edge (u, v):
            if v has a guess:
                Set v.guess = min(v.guess, u.dist + w(u, v))
            else
                Set v.guess = u.dist + w(u, v)
```

Correctness? Maintain invariant:

- All set dist are correct
- All set guess are length of shortest path *that only uses intermediate vertices that have already had dist set*

Proof of Correctness

Desired Invariant

- All set `dist` are correct
- All set `guess` are length of shortest path *that only uses intermediate vertices that have already had* `dist` *set*

Holds at the beginning: no `dist` set, only `guess` is $\mathbf{0}$ for s.

Proof of Correctness

Desired Invariant

- All set `dist` are correct
- All set `guess` are length of shortest path *that only uses intermediate vertices that have already had* `dist` *set*

Holds at the beginning: no `dist` set, only `guess` is $\mathbf{0}$ for s.

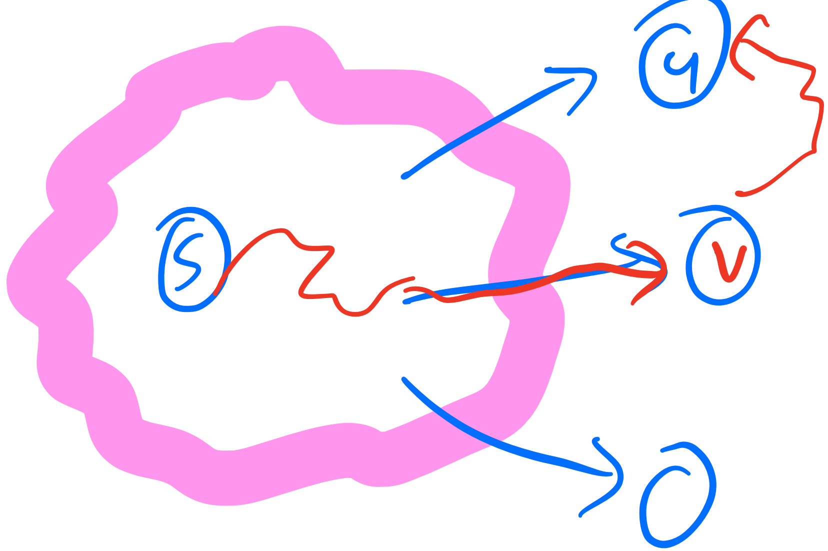
If it holds at the end, the values of `dist` we return are correct!

Proof of Correctness II

Desired Invariant

- All set `dist` are correct
- All set `guess` are length of shortest path *that only uses intermediate vertices that have already had* `dist` *set*

Suppose the invariant currently holds. Why is the smallest `guess` guaranteed to be a correct `dist`?



$\text{len}(\rightsquigarrow) \geq$

$v.\text{guess}$

$\geq u.\text{guess}$

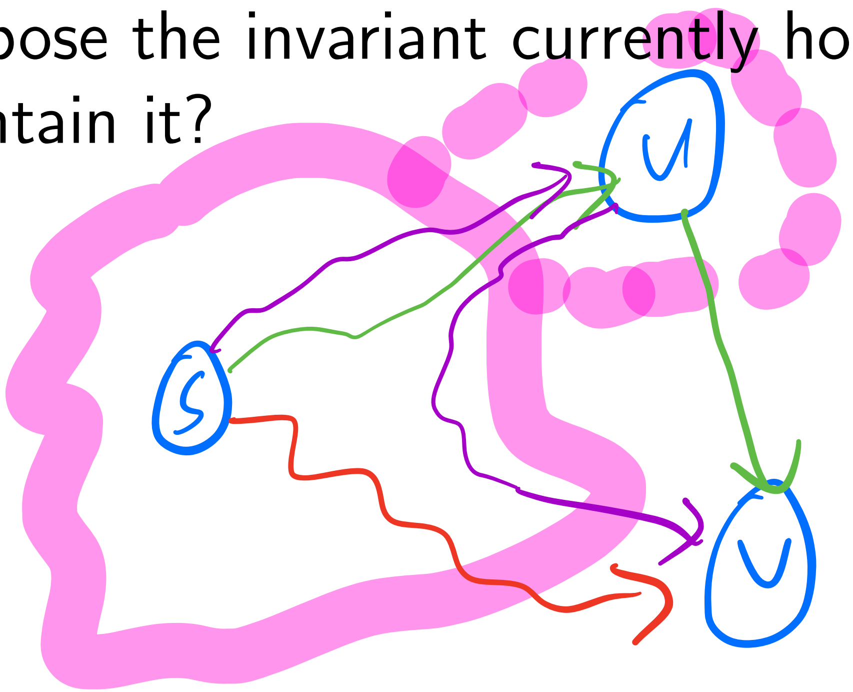
$\Rightarrow \text{dist}(s,u) = u.\text{guess}$

Proof of Correctness III

Desired Invariant

- All set `dist` are correct
- Set `guess` to length of shortest path *that only uses intermediate vertices that have already had* `dist` *set*

Suppose the invariant currently holds. Why do our updates to guess maintain it?



previous guess

u.dist + w(u,v)

never shortest

v.guess = min(⇝ , ⇝)

Efficiency of Intuitive Algorithm

```
IntuitionWSSSP(G, s):
    Set s.guess = 0
    while there exists a vertex with a guess but no dist:
        Let u be such a vertex with the smallest guess
        Set u.dist = u.guess
        for each edge (u, v):
            if v has a guess:
                Set v.guess = min(v.guess, u.dist + w(u, v))
            else
                Set v.guess = u.dist + w(u, v)
```

repeat V times

O(V)

repeat Once per edge

O(E)

Efficiency? $O(V^2 + E)$

Efficiency of Intuitive Algorithm

```
IntuitionWSSSP(G, s):
    Set s.guess = 0
    while there exists a vertex with a guess but no dist:
        Let u be such a vertex with the smallest guess
        Set u.dist = u.guess
        for each edge (u, v):
            if v has a guess:
                Set v.guess = min(v.guess, u.dist + w(u, v))
            else
                Set v.guess = u.dist + w(u, v)
```

Efficiency? $O(V^2 + E)$

Can we do better?

Priority Queues

Inefficiency: spend a lot of time looking for the smallest `guess`.

Priority Queues

Inefficiency: spend a lot of time looking for the smallest `guess`.

This is exactly what *priority* queues are designed for!

- `Insert`(v, k): insert v with key k
- `DecreaseKey`(v, k): decrease v's key to k
- `ExtractMin`$()$: remove and return v with smallest key

Dijkstra's Algorithm

```
Dijkstra(G, s):
    Set s.dist = 0 and v.dist = ∞ for all v ≠ s
    Insert(v, v.dist) for all vertices v
    while the priority queue is non-empty:
        u = ExtractMin()
        for each edge (u, v):
            if u.dist + w(u, v) < v.dist:
                Set v.dist = u.dist + w(u, v)
                DecreaseKey(v, v.dist)
```

Dijkstra's Algorithm

```
Dijkstra(G, s):
    Set s.dist = 0 and v.dist = ∞ for all v ≠ s
    Insert(v, v.dist) for all vertices v → V times
    while the priority queue is non-empty:
        u = ExtractMin() → V times
        for each edge (u, v):
            if u.dist + w(u, v) < v.dist:
                Set v.dist = u.dist + w(u, v)
                DecreaseKey(v, v.dist) → O(E) times
```

Efficiency?

Dijkstra's Algorithm

```
Dijkstra(G, s):
    Set s.dist = 0 and v.dist = ∞ for all v ≠ s
    Insert(v, v.dist) for all vertices v
    while the priority queue is non-empty:
        u = ExtractMin()
        for each edge (u, v):
            if u.dist + w(u, v) < v.dist:
                Set v.dist = u.dist + w(u, v)
                DecreaseKey(v, v.dist)
```

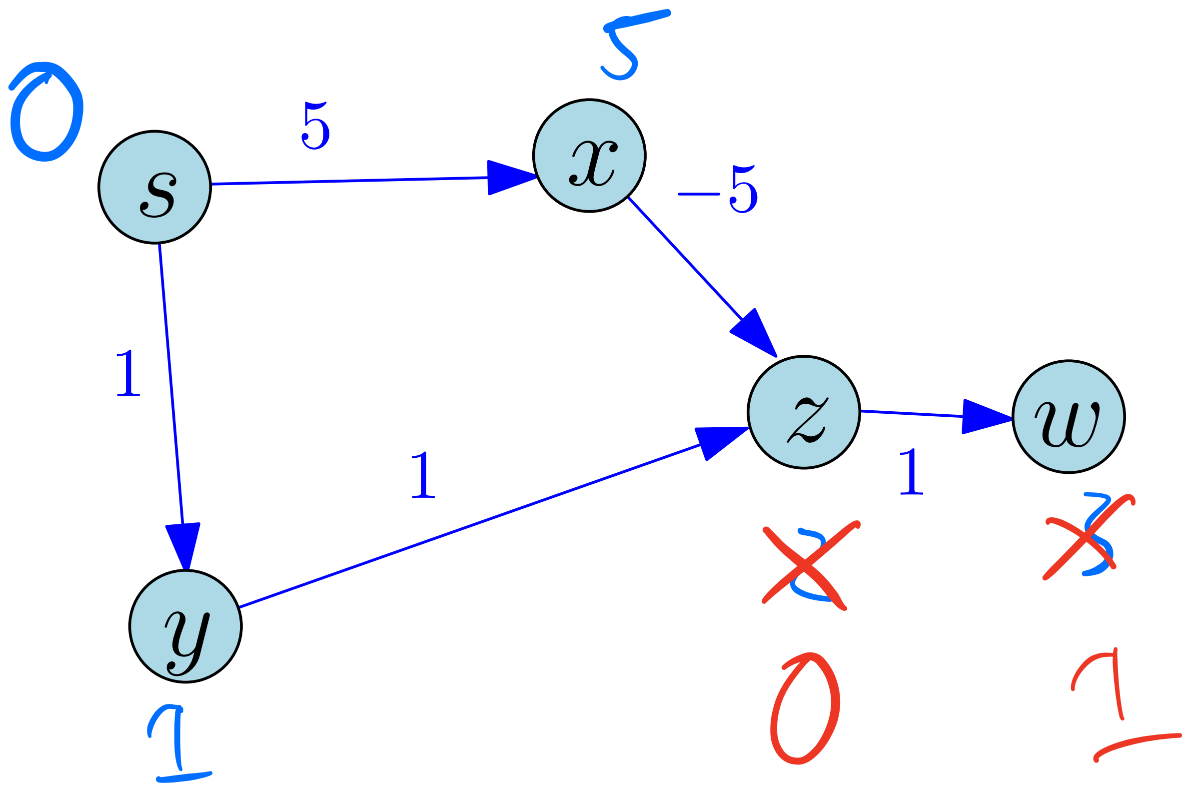
Efficiency?

`Insert` and `ExtractMin` $O(V)$ times, `DecreaseKey` $O(E)$ times

Standard implementation runs each operation in $O(\log V)$ time.

$O((V+E)\log V)$

A Counterexample :(



Dijkstra and Negative Weights

Proof of correctness depends on the assumption that adding more edges to a path can only make it longer.

Dijkstra and Negative Weights

Proof of correctness depends on the assumption that adding more edges to a path can only make it longer.

This is only true if and only if all edges have non-negative weights!

Dijkstra and Negative Weights

Proof of correctness depends on the assumption that adding more edges to a path can only make it longer.

This is only true if and only if all edges have non-negative weights!

Takeaway: Dijkstra can solve SSSP *with non-negative weights* in $O((V+E)\log V)$ time, but we need a different approach to deal with graphs that have negative edge weights.

Part IV

Negative Weights

Why Negative Weights?

Most problems you would intuitively think of as shortest path problems involve spending some resource (eg time, distance, etc).

Why Negative Weights?

Most problems you would intuitively think of as shortest path problems involve spending some resource (eg time, distance, etc).

Most of the time non-negative weights are enough, but you will still sometimes run into a setting where negative weights are "natural".

Why Negative Weights?

Most problems you would intuitively think of as shortest path problems involve spending some resource (eg time, distance, etc).

Most of the time non-negative weights are enough, but you will still sometimes run into a setting where negative weights are “natural”.

Example: given a list of currency exchange rates, what is the most efficient way to covert currency i into currency j?

	Dollar	Rupee	Yen
1 Dollar:	X	80	150
1 Rupee:	1/85	X	2
1 Yen:	1/165	2/5	X

1 Dollar → 150 Yen

1 Dollar → 80 Rupee

→ 160 Yen

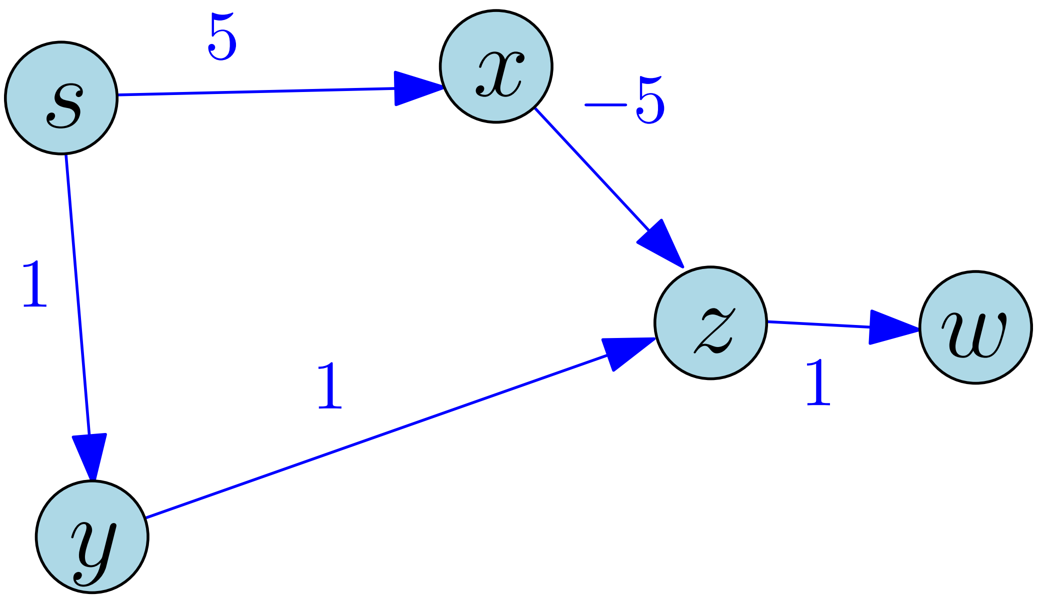
Currency Reduction

Starting point: what is the value we get from conversions through currencies $c_1, c_2, \ldots, c_k$? (Say E_{ij} is exchange rate of i to j)

How do we make this into a (standard) shortest-path problem?

Dealing with Negative Weights

Intuitively, Dijkstra's fails on negative edge weights because it "locks in" distances when we may later find an even shorter path.



Dealing with Negative Weights

Intuitively, Dijkstra's fails on negative edge weights because it "locks in" distances when we may later find an even shorter path.

What happens if we don't "lock in" distances until the end?

Dealing with Negative Weights

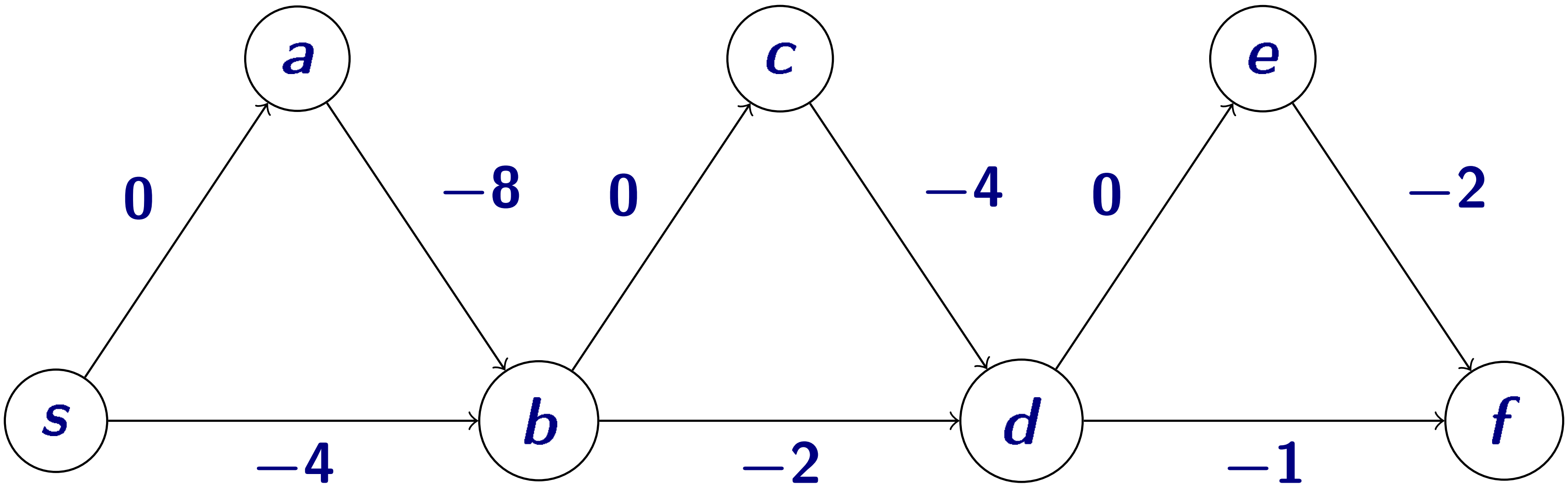
Intuitively, Dijkstra's fails on negative edge weights because it "locks in" distances when we may later find an even shorter path.

What happens if we don't "lock in" distances until the end?

```
NegativeDijkstra(G, s):
    Set s.dist = 0 and v.dist = ∞ for all v ≠ s
    Insert(v, v.dist) for all vertices v
    while the priority queue is non-empty:
        u = ExtractMin()
        for each edge (u, v):
            if u.dist + w(u, v) < v.dist:
                Set v.dist = u.dist + w(u, v)
                if v is in the queue:  DecreaseKey(v, v.dist)
                else Insert(v, v.dist)
```

An Adversarial Example

What happens to `NegativeDijkstra` on the following graph?



Negative Weights Takeaways

Dijkstra's can be made to work with negative edge weights, but the run time is exponential in the worst case!

The priority queue can be "tricked" into making many unnecessary updates; we need a better method to determine which vertex to process next. (Next time!)

Problems and Algorithms From Today

- Shortest paths from s in an unweighted graph
  - BFS, $O(V + E)$ time

- Shortest paths from s in a non-negative weighted graph
  - Dijkstra's, $O((V + E) \log V)$