Objectives

Greedy Algorithms

- Optimal caching
- Shortest path

Optimal Offline Caching: Farthest-In-Future

Evict item in cache that is not requested until farthest in the future

current cache: a b c d e f

future queries: g a b c e d a b b a c d e a f a d e f g h ...

cache miss — eject this one

Theorem. [Bellady, 1960s] FF is optimal eviction schedule

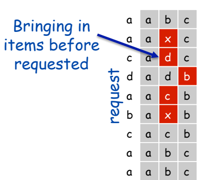
Pf. Algorithm and theorem are intuitive; proof is subtle

- Better than least frequently used?

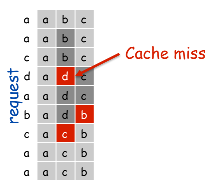
Reduced Eviction Schedules

Def. A reduced schedule is a schedule that only inserts an item into the cache when that item is requested

- No bringing in an item ahead of time; minimal amt of work per step



an unreduced schedule



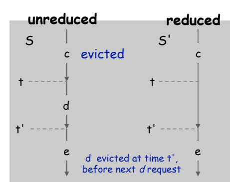
a reduced schedule

Reduced Eviction Schedules

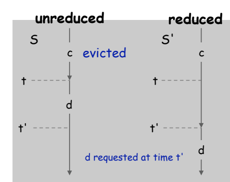
Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more cache misses

Pf. (by induction on number of unreduced items)

- Case 1: *d* evicted at time t', before next request for *d*
- Case 2: *d* requested at time t' before *d* is evicted



Case 1



Case 2

Farthest-In-Future: Analysis

Theorem. FF is optimal eviction algorithm

Pf Sketch

- Let S_{FF} be schedule by Farthest-in-Future
- Let S* be optimal schedule
  - Fewest possible cache misses
- Transform S* into S_{FF}
  - One eviction decision at a time
  - Not increasing number of cache misses

Farthest-In-Future: Analysis

Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FF} through the first j+1 requests.

Let S be reduced schedule that satisfies invariant through *j* requests. We produce reduced schedule S' that satisfies invariant after j+1 requests

- Consider (j+1)st request d = d_{j+1}
- Since S and S_{FF} have agreed up until now, they have *same cache contents* before request j+1
- What are the possibilities for what happens on (j+1)st request?

Farthest-In-Future: Analysis

Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FF} through the first j requests.

Let S be reduced schedule that satisfies invariant through *j* requests. We produce S' that satisfies invariant after *j+1* requests.

- Consider $(j+1)^{st}$ request $d = d_{j+1}$
- Since S and S_{FF} have agreed up until now, they have the same cache contents before request j+1
- Case 1: d is already in the cache. S' = S satisfies invariant
- Case 2: d is not in the cache and S and S_{FF} evict the same element.
  S' = S satisfies invariant.

Farthest-In-Future: Analysis

Pf. (continued)

- Case 3: d is not in the cache; S_{FF} evicts e; S evicts f ≠ e
  - begin construction of S' from S by evicting e instead of f

	S			S'		
j	same	e	f	same	e	f
j+1	same	e	d	same	d	f

  - now S' agrees with S_{FF} on first *j* requests; we show that having element *f* in cache is *no worse* than having element *e*
    - Need to get schedules' caches back in sync again
    - All decisions will be the same until decision involves *e* or *f*

Farthest-In-Future: Analysis

Let j' be the first time after j+1 that S and S' take a *different* action, and let g be item requested at time j'.

must involve e or f (or both)

	S		S'	
j'	same	e	same	f

- What are the possibilities for *g*?
  - Is *g* in the cache for S? For S'?
  - What does their caches look like afterwards?

Farthest-In-Future: Analysis

Let j' be the first time after j+1 that S and S' take a *different* action, and let g be item requested at time j'.

	S		S'	
j'	same	e	same	f

- Case 3a: g = e
  - Can't happen with Farthest-In-Future since there must be a request for *f* before *e*

Farthest-In-Future: Analysis

Let j' be the first time after j+1 that S and S' take a *different* action, and let g be item requested at time j'

	S		S'	
j'	same	e	same	f

- Case 3b: g ≠ e, f
  - g is not in either cache
  - S must evict e
    - otherwise S' would take the same action
  - Make S' evict f; now S and S' have the same cache:

	S		S'	
j'	same	g	same	g

Farthest-In-Future: Analysis

Let j' be the first time after j+1 that S and S' take a *different* action, and let g be item requested at time j'.

	S		S'	
j'	same	e	same	f

- Case 3c: g = f
  - Element f can't be in cache of S, so let e' be the element that S evicts
    - If e' = e, now S and S' have same cache
    - If e' ≠ e, S' evicts e' and brings e into the cache; now S and S' have the same cache

Note: S' is no longer reduced, but can be transformed into a reduced schedule that agrees with S_{FF} through step j+1

Farthest-In-Future: Analysis

Let j' be the first time after j+1 that S and S' take a *different* action, and let g be item requested at time j'.

j'	same	e		same	f
	S			S'	

For both cases (3b, 3c), have reduced schedule S' that agrees with S_{FF} for first j+1 items

Farthest-in-Future: Analysis

Theorem. FF is optimal eviction algorithm

Pf. (by induction on number of requests j)

Let S* be an optimal schedule

Construct an optimal schedule S_1 that agrees with S_{FF} through the first step

Apply previous proof inductively for j = 1, 2, 3, …, m, producing schedules S_j that agree with S_{FF} through first *j* steps

Each schedule S_j incurs no more misses than the corresponding S_{FF} one

$S_m = S_{FF}$ because agrees through whole sequence

Caching Perspective

Online vs. offline algorithms

- Offline: full sequence of requests is known a priori
- Online (reality): requests are not known in advance
- Caching is among most fundamental online problems in CS

LIFO. Evict page brought in most recently

LRU. Evict page whose most recent access was earliest

Theorem. FF is optimal *offline* eviction algorithm

FF with direction of time reversed!

- Provides basis for understanding and analyzing online algorithms.
- LRU is k-competitive. [Section 13.8]
- LIFO is arbitrarily bad

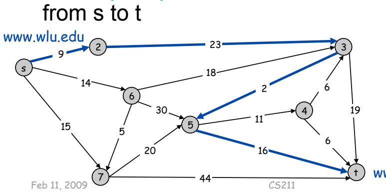
SHORTEST PATHS IN A GRAPH

Shortest Path Problem

Given

- Directed graph G = (V, E)
- Source s, destination t
- Length ℓ_e = length of edge e (non-negative)

Shortest path problem: find shortest directed path from s to t



cost of path = sum of edge costs in path

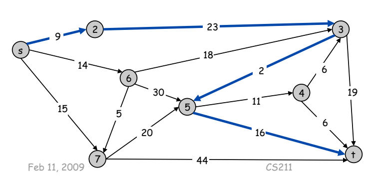
Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48

Shortest Path Problem

Shortest path problem: find shortest directed path from s to t

Towards algorithm ideas:

- What is shortest path from s to 2? To 6?
- What is the shortest path to 3? 5? 7?



cost of path = sum of edge costs in path

Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48

Dijkstra's Algorithm

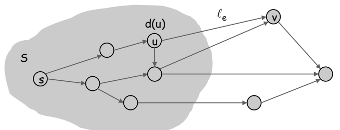
Maintain a set of explored nodes S

- Know the shortest path distance d(u) from *s* to *u*

Initialize S={s}, d(s)=0

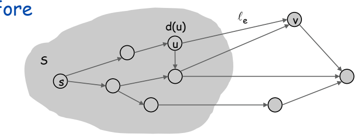
Repeatedly choose unexplored node *v* which minimizes $\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e,$ shortest path to some u in explored part, followed by a single edge (u, v)

- add v to S and set d(v) = π(v)

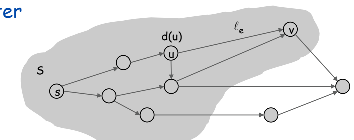


Dijkstra's Algorithm

Before

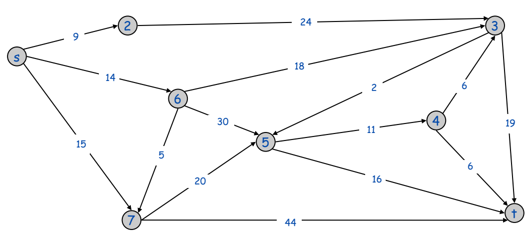


After



Dijkstra's Shortest Path Algorithm

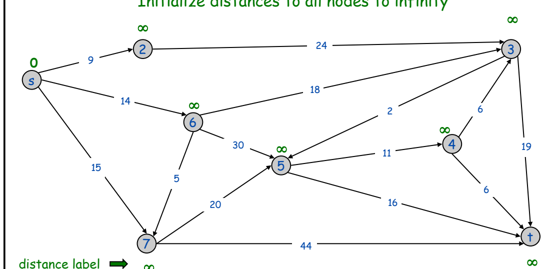
Find shortest path from s to t.



Dijkstra's Shortest Path Algorithm

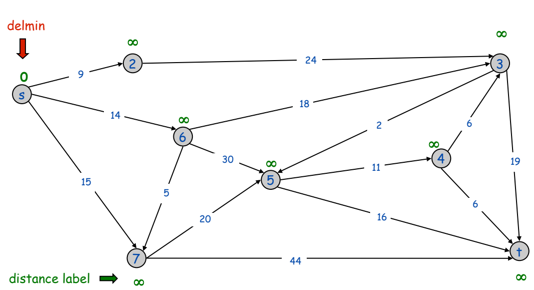
S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }

Initialize distances to all nodes to infinity



Dijkstra's Shortest Path Algorithm

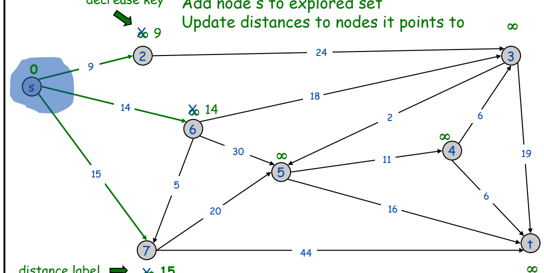
S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

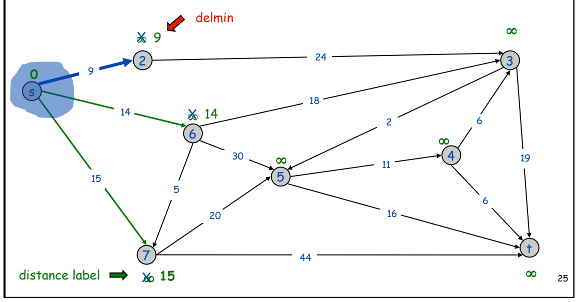
S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }

Add node s to explored set
Update distances to nodes it points to



Dijkstra's Shortest Path Algorithm

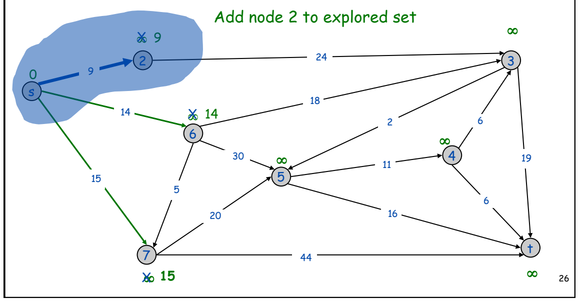
S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



25

Dijkstra's Shortest Path Algorithm

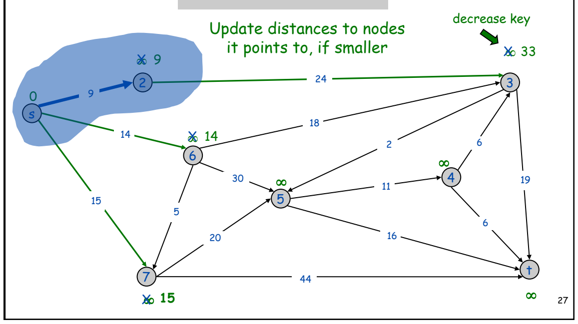
S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



26

Dijkstra's Shortest Path Algorithm

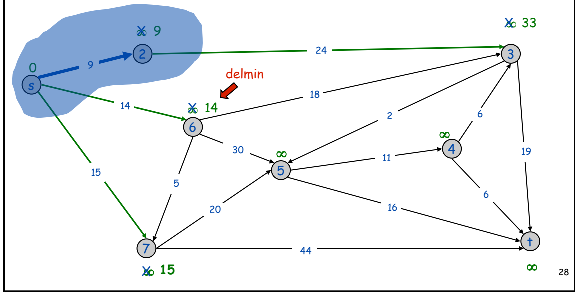
S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



27

Dijkstra's Shortest Path Algorithm

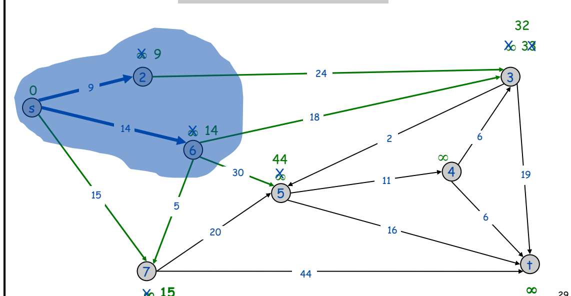
S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



28

Dijkstra's Shortest Path Algorithm

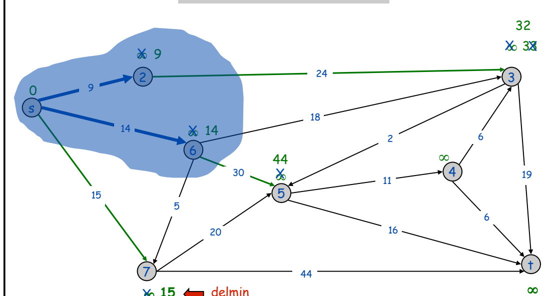
S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



29

Dijkstra's Shortest Path Algorithm

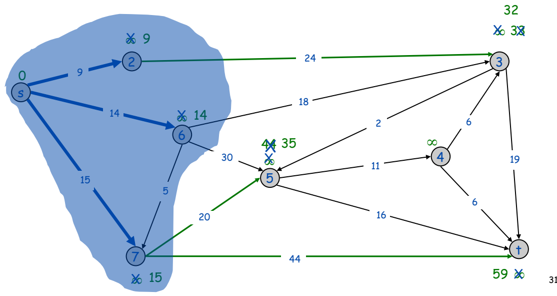
S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



30

Dijkstra's Shortest Path Algorithm

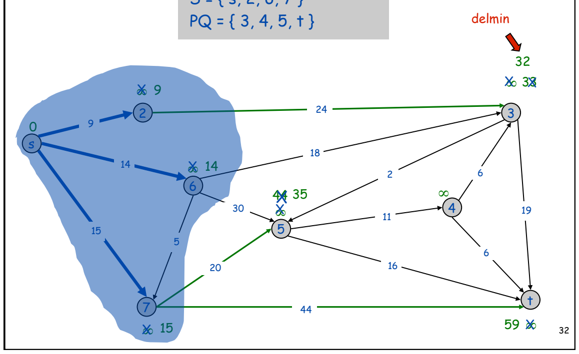
S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



31

Dijkstra's Shortest Path Algorithm

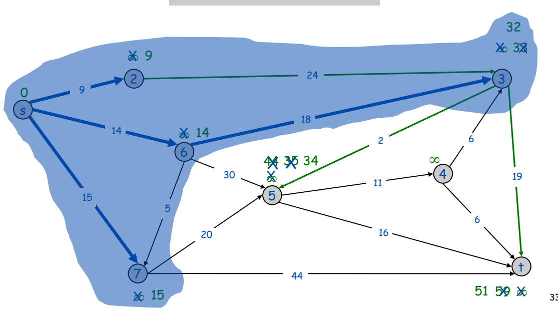
S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



32

Dijkstra's Shortest Path Algorithm

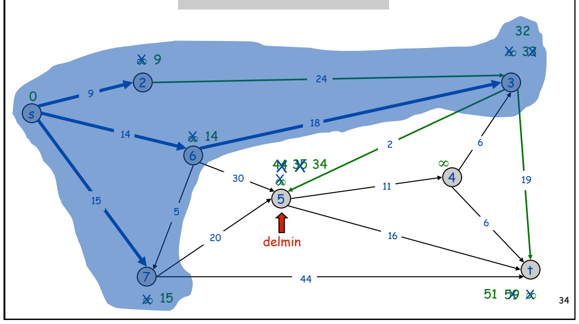
S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



33

Dijkstra's Shortest Path Algorithm

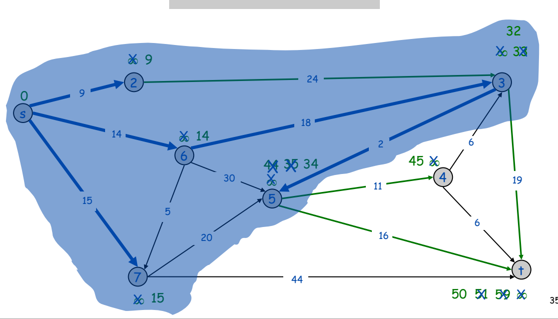
S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



34

Dijkstra's Shortest Path Algorithm

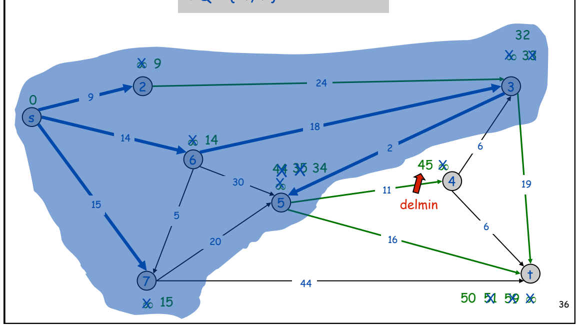
S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



35

Dijkstra's Shortest Path Algorithm

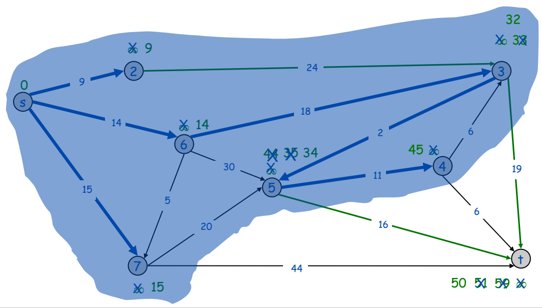
S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



36

Dijkstra's Shortest Path Algorithm

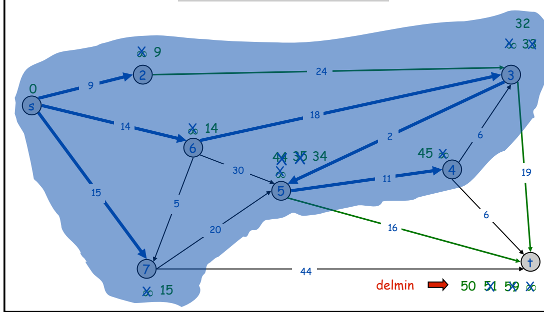
S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



37

Dijkstra's Shortest Path Algorithm

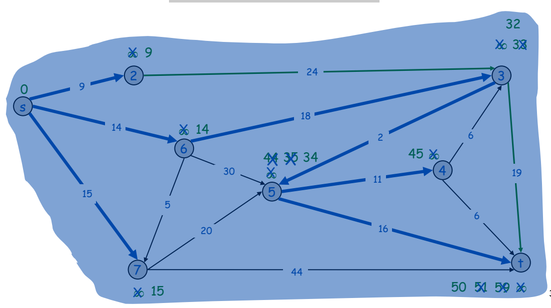
S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



38

Dijkstra's Shortest Path Algorithm

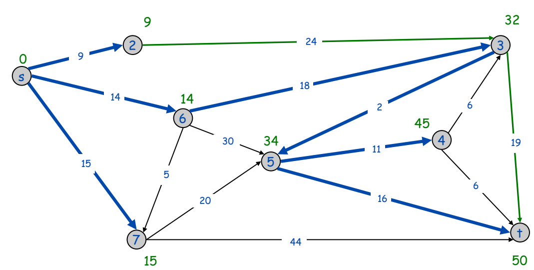
S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



39

Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



40

Dijkstra's Algorithm

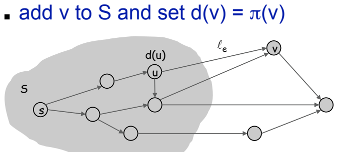
Maintain a set of explored nodes S

- Know the shortest path distance d(u) from *s* to *u*

Initialize S={s}, d(s)=0

Repeatedly choose unexplored node *v* which minimizes $\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e,$ ← shortest path to some u in explored part, followed by a single edge (u, v)

- add v to S and set d(v) = π(v)



Running time?
Implementation?
Data structures?

Dijkstra's Algorithm

Maintain a set of explored nodes S

- Know the shortest path distance d(u) from *s* to *u*

Initialize S={s}, d(s)=0

Repeatedly choose unexplored node *v* which minimizes $\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e,$ ← shortest path to some u in explored part, followed by a single edge (u, v)

- add v to S and set d(v) = π(v)

Using a priority queue, how many
- Inserts?
- Finding minimum?
- Deletions?
- Updating the key?
- Determining if empty?

How long does each operation take?

Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain

$$\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e \,.$$

- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update $\pi(w) = \min\{\, \pi(w),\ \pi(v) + \ell_e \,\}$.

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$

PQ Operation	Dijkstra	Binary heap
Insert	n	log n
ExtractMin	n	log n
ChangeKey	m	log n
IsEmpty	n	1
Total		m log n

How Greedy?

How Greedy?

We always form shortest new *s-v* path from a path in S followed by a *single* edge

Proof of optimality: *Stays ahead* of all other solutions

- Each time selects a path to a node *v*, that path is shorter than every other possible path to *v*

Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node u ∈ S, d(u) is the length of the shortest s-u path

Pf. (by induction on |S|)

Base case: |S|=1 …

Inductive hypothesis?

Next step?

Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node u ∈ S, d(u) is the length of the shortest s-u path

Pf. (by induction on |S|)

Base case: For |S| = 1, S={s}; d(s) = 0

Inductive hypothesis: Assume true for |S| = k, k ≥ 1

- Grow |S| to k+1
- Adding next node *v* by *u*→ *v*
- What do we know about *s*→ *u*?
- What can we say about other *s*→ *v* paths?
- Why didn't we pick *y* as the next node?