Parallel And Sequential Data Structures and Algorithms

Shortest Paths II: Bellman-Ford and Floyd-Warshall

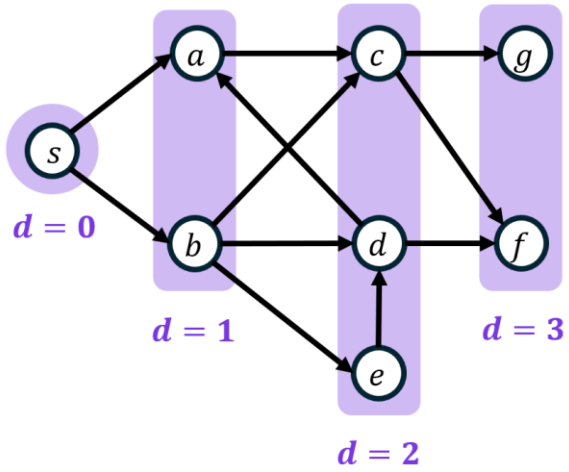
Announcements

- **TA applications** are open!
  - https://edstem.org/us/courses/102226/discussion/8479987
- **Midterm Two grades** will be out soon

Learning Objectives

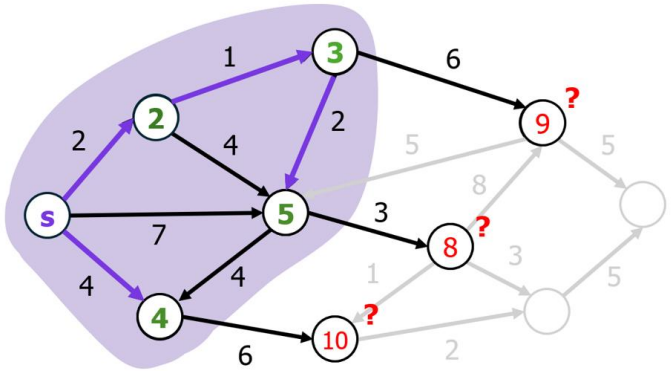
- Algorithms for shortest paths on graphs with negative weights
  - Single-source shortest paths on directed **acyclic** graphs
  - Single-source shortest paths with negative weights (**Bellman-Ford**)
  - All-pairs shortest paths with negative weights (**Floyd-Warshall**)
- See how to detect the presence of **negative-weight cycles**, which may make shortest paths undefined

Shortest Path Algorithms



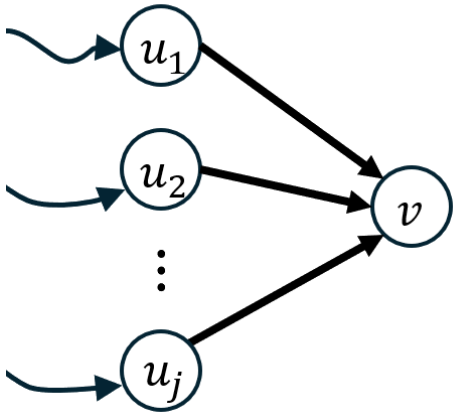
BFS

Shortest Paths on **Unweighted** Graphs



Dijkstra's

Shortest Paths on **Non-negative-Weighted** Graphs

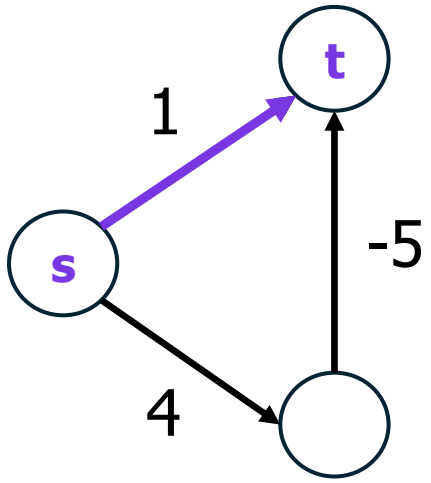


Bellman-Ford & Floyd-Warshall

Shortest Paths on **Negative-Weighted** Graphs!

Negative Weights and Dijkstra

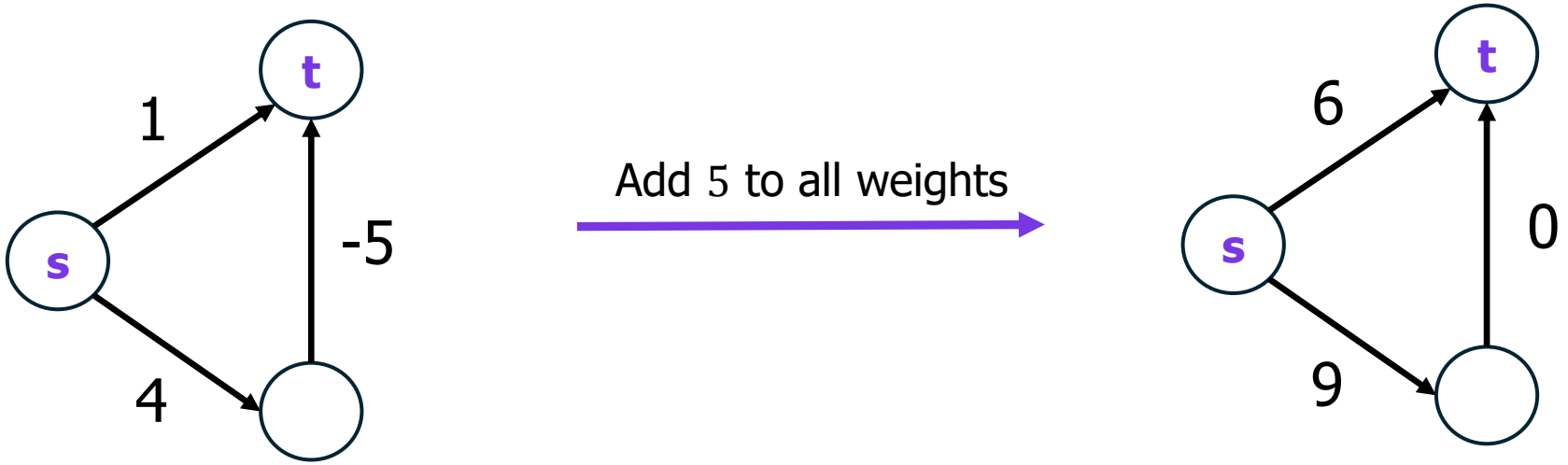
What will Dijkstra's Algorithm do on this graph with negative weights?



Answer: Greedily selects the edge (s, t) with weight 1, which misses the path with lower weight -1.

Negative Weights and Dijkstra

- Can we **modify** the graph to make it work?



No. Adding a constant to all edges **makes longer paths disproportionately heavier**; it breaks the ordering of the paths, so the shortest path is no longer the shortest path.

Negative-Weight Cycles

In the presence of a **negative-weight cycle**, a shortest s-t path **may** not exist (if the cycle is reachable from s and can reach t)

- A shortest-path algorithm for non-negative weights will always fine a valid answer (provided that s can reach t)
- For negative weights, we have a choice:
  - Detect a negative-weight cycle and report invalid paths
  - Just silently return bogus answers
- We will discuss how to detect negative cycles

The Sub-Paths Property

Shortest Paths with Negative Weights

Can we find some **useful property** of shortest paths to exploit?

- Consider a directed graph and a source vertex s
- Consider a shortest $s \to v$ path for some vertex v



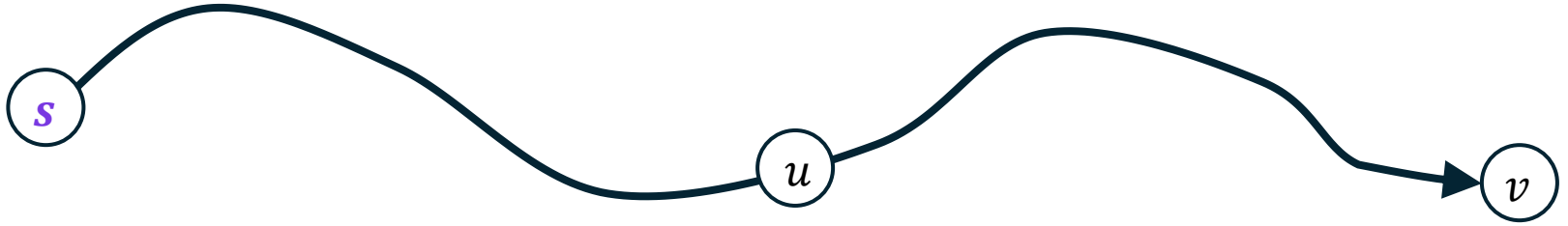
? **What must be true** about vertex u, the second-last vertex on the path?

The **sub-path** from s to u is itself a shortest path!

Shortest Paths with Negative Weights

Can we find some **useful property** of shortest paths to exploit?

- Consider a directed graph and a source vertex s
- Consider a shortest $s \to v$ path for some vertex v

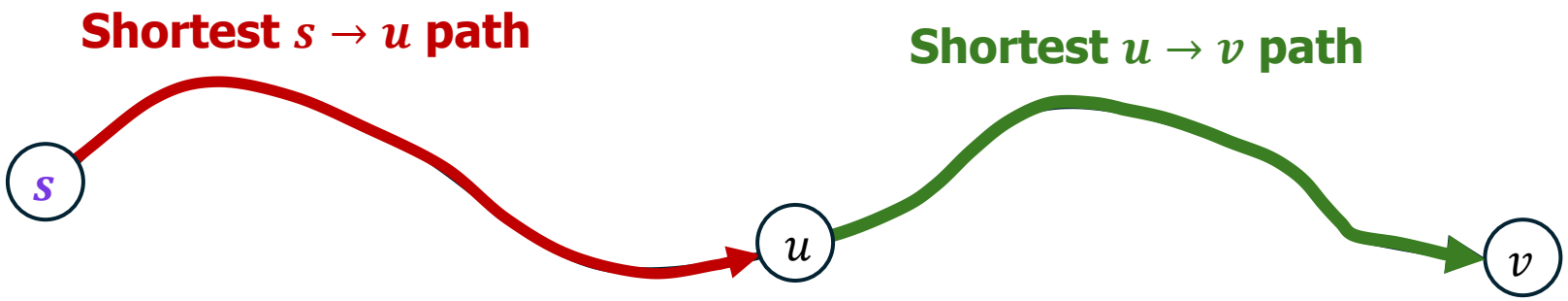


? What if u is **any vertex** on a shortest $s \to v$ path?

The **sub-paths** from s to u and from u to v are **both** shortest paths!

The sub-paths property

Theorem (Sub-paths property): Any sub-path of a shortest path is itself also a shortest path between its endpoints

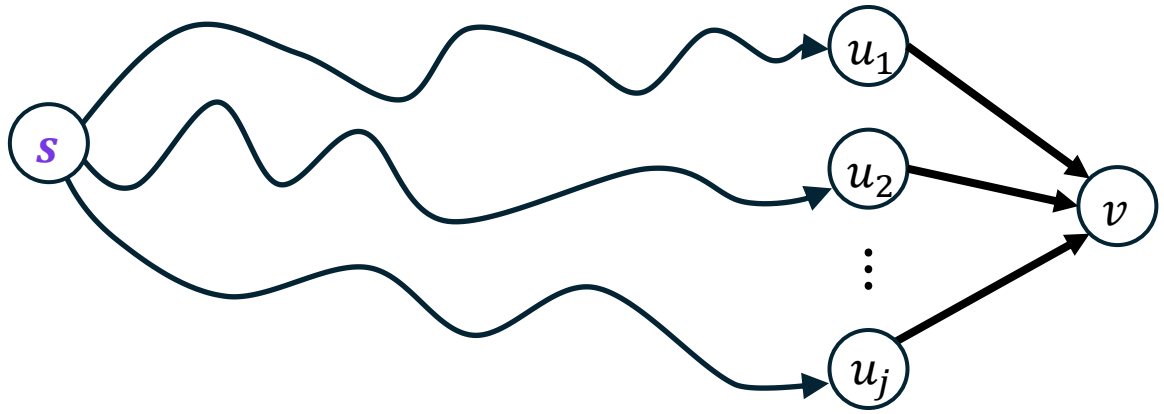


This property is true regardless of whether there are **negative weights**!

Shortest Paths with Negative Weights

Shortest Paths with Negative Weights

- Consider a directed graph and a source vertex s
- Consider a vertex v who's **in-neighbours** are $u_1, u_2, \ldots, u_j$

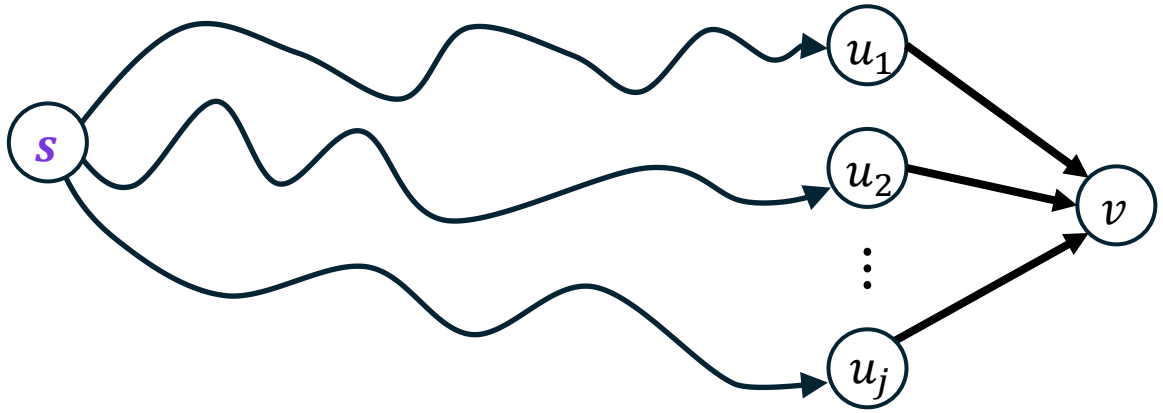


? What does the sub-paths property say about the shortest path weight to v?

The shortest path to v is the **minimum over its in-neighbours**

Shortest Paths with Negative Weights

- Consider a directed graph and a source vertex s
- Consider a vertex v who's **in-neighbours** are $u_1, u_2, \ldots, u_j$

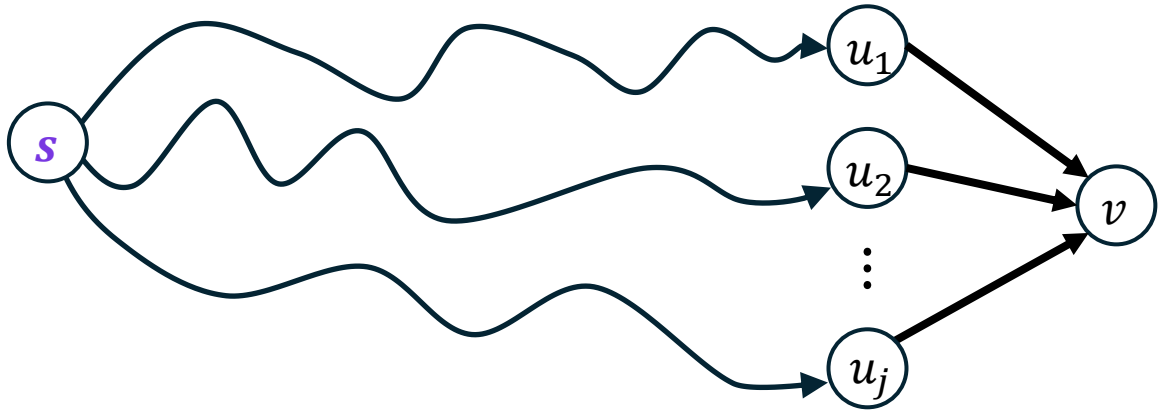


$$\delta(s,v) = \min \begin{cases} \delta(s,u_1) + w(u_1,v) \\ \delta(s,u_2) + w(u_2,v) \\ \vdots \\ \delta(s,u_j) + w(u_j,v) \end{cases}$$

We can express this as a **recurrence relation**

Shortest Paths with Negative Weights

- Consider a directed graph and a source vertex s
- Consider a vertex v who's **in-neighbours** are $u_1, u_2, \ldots, u_j$



$$\delta(s,v) = \min \begin{cases} \delta(s,u_1) + w(u_1,v) \\ \delta(s,u_2) + w(u_2,v) \\ \vdots \\ \delta(s,u_j) + w(u_j,v) \end{cases}$$

In general, $\delta(s,v) = \min_{u \in N^-(v)} \big(\delta(s,u) + w(u,v)\big)$

Shortest Paths as a Recurrence

In general, $\delta(s,v) = \min_{u \in N^-(v)} \big(\delta(s,u) + w(u,v)\big)$

? Can we solve for shortest path weights by **evaluating** this recurrence?

- You can't evaluate it because trying to evaluate the recursion would cycle forever (infinite recursion!)

? Is there a **specific class of graph** on which we can evaluate it?

- We can evaluate it on **acyclic graphs!**

Evaluating the Recurrence

- Let $\mathrm{Dist}(v)$ denote the weight of the shortest path from s to v for all $v \in V$. Then consider the following recurrence

$$\mathrm{Dist}(v) = \begin{cases} 0, & \text{if } v = s \\ \min\limits_{u \in N^-(v)} \big(\mathrm{Dist}(u) + w(u,v)\big), & \text{otherwise} \end{cases}$$

- Evaluating this naively could take exponential time…

Use **dynamic programming**! This gives an efficient algorithm for all-pairs shortest paths on an acyclic graph with negative weights

Cost Analysis

Theorem (SSSP in a DAG): On a directed acyclic graph on n vertices and m edges, shortest paths can be found in $O(n+m)$ cost

Proof: There are n subproblems (one per vertex)

- *For each subproblem, the cost of evaluating it is the in-degree of that vertex, i.e., $|N^-(v)|$*
- *Therefore, the sum of costs of every subproblem is the sum of the in-degrees in the graph*
- *Each in-degree comes from one edge, so that cost is $O(m)$*
- *Therefore, the total cost is $O(n+m)$*

k-hop Distances and Bellman-Ford

Weighted Graphs with Cycles

- The following recurrence worked for acyclic graphs, but not cyclic graphs because it would cycle forever and never reach a base case!

$$\text{Dist}(v) = \begin{cases} 0, & \text{if } v = s \\ \min_{u \in N^-(v)} \big(\text{Dist}(u) + w(u,v)\big), & \text{otherwise} \end{cases}$$

? Can we add information (i.e., **strengthen it**) to make it not cyclic?

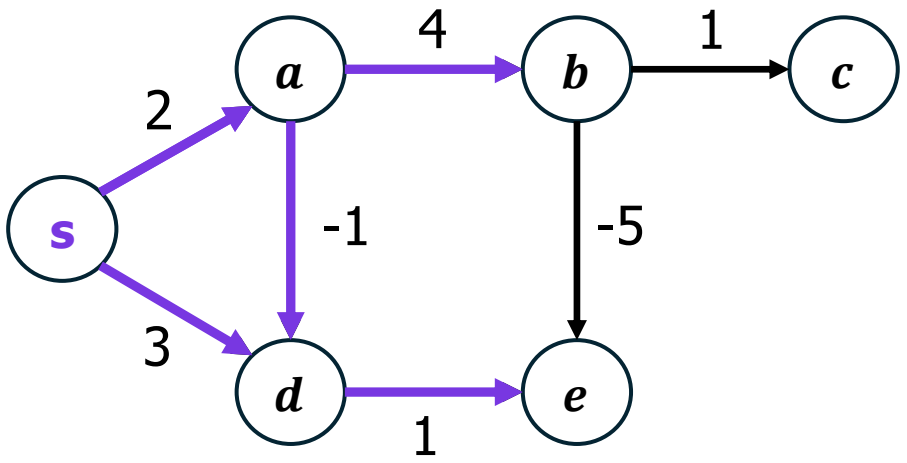
Hint: What else is true about the $s \to u$ path from the sub-paths property?



The sub-path of a shortest path consisting of k **edges** is a path consisting of $k-1$ **edges**!

k-hop Shortest Paths

Definition (k-hop Distance): A ***k*-hop shortest path** between u and v is a shortest path from u to v that contains at most k edges. The ***k*-hop distance** from u to v is the weight of a k-hop shortest path.



Vertex	2-hop dist
s	0
a	2
b	6
c	∞
d	1
e	4

From k-hop to the answer

- How do k-hop distances help us find shortest paths?

Theorem (Maximum Hops): In a graph with no negative-weight cycles, the shortest path between any vertices u and v uses at most $n - 1$ edges.

(Last lecture we argued that a shortest path needs no more than $n - 1$ edges)

Corollary (Enough Hops): In a graph with no negative-weight cycles, the shortest path distances are the $(n - 1)$-hop distances.

Dynamic Programming for k-hop

- We can modify our recurrence for acyclic graphs to solve the k-hop shortest path problem by introducing a k parameter:

$$\mathbf{Dist}(v, k) = \text{The } k\text{-hop distance from } s \text{ to } v$$

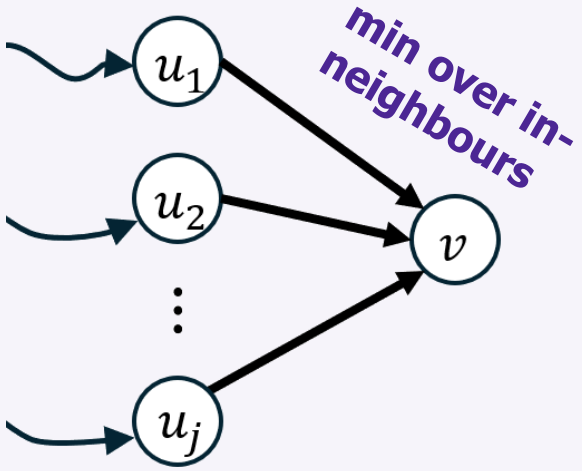
$$\mathbf{Dist}(v, \boldsymbol{k}) = \begin{cases} 0, & \text{if } \boldsymbol{k = 0} \text{ and } v = s \\ \infty, & \text{if } \boldsymbol{k = 0} \text{ and } \boldsymbol{v \neq s} \\ \min\left(\begin{cases} \mathbf{Dist}(v, k-1) \\ \min\limits_{u \in N^-(v)} \left(\mathrm{Dist}(u, \boldsymbol{k-1}) + w(u, v) \right) \end{cases} \right), & \text{otherwise} \end{cases}$$

Interesting subtlety: Evaluating this requires the **in-neighbours** $N_G^-(v)$

k-hop Pseudocode

```
// Requires a reverse adjacency list representation
type RevAdjList = sequence<sequence<(int,real)>>

fun khop_dists(in_adj : RevAdjList, source: int, K : int) -> sequence<real>:
  n = |adj|
  dist = [[∞ for _ in 0...K] for _ in 0...n-1]
  dist[source][0] ← 0
  for k in 1...K:
    for v in 0...n-1:
      dist[v][k] ← dist[v][k-1]
      for u, w in in_adj[v]:
        dist[v][k] ← min(dist[v][k], dist[u][k-1] + w)
  return dist
```



Negative Cycles

- We know that in a graph with no negative cycles, the $(n-1)$-hop paths are the shortest paths
- How do we detect a negative cycle rather than returning bogus shortest path distances?

Theorem (Identifying Negative Cycles): There exists a negative weight cycle reachable from s **if and only if** there exists a vertex v such that

$$\mathbf{Dist}(v, n) < \mathbf{Dist}(v, n-1)$$

So, to detect the existence of a **negative cycle**: compute the n-hop shortest paths and see if they differ from the $(n-1)$-hop shortest paths!

The Bellman-Ford Algorithm

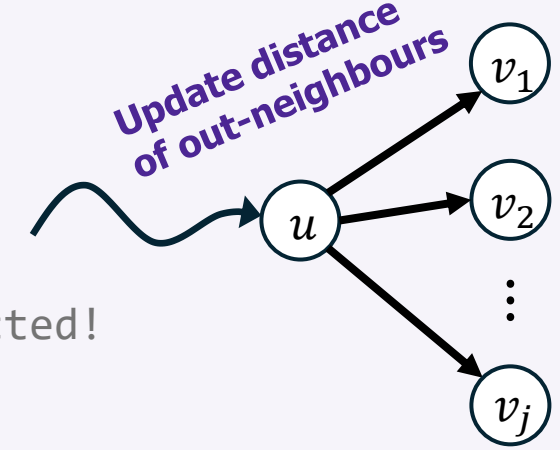
- Requiring a reverse (in-neighbour) adjacency list is a bit odd.
- We can reorder the k-hop distances computation to make it work on an ordinary (out-neighbour) adjacency list instead
- We also don't need to store k-hop distances for all values of k. We only need the current value of k
- Lastly, we check for the existence of a negative cycle by checking whether any distance changes when $k = n$

Making these changes to the k-hop distances algorithm results in the **Bellman-Ford** single-source shortest paths algorithm.

The Bellman-Ford Algorithm

```
// Assume a directed adjacency list representation
type WeightedAdjList = sequence<sequence<(int,real)>>

fun bellman_ford(adj : WeightedAdjList, source : int) -> option<sequence<real>>:
  n = |adj|
  dist = [∞ for _ in 0...n-1]
  dist[source] = 0
  for k in 1...n:
    for u in 0...n-1:
      for v, w in adj[u]:
        if dist[u] + w < dist[v]:
          if k == n: return NONE  // negative cycle detected!
          dist[v] ← dist[u] + w
  return SOME(dist)
```



The Bellman-Ford Algorithm

```
// Assume a directed adjacency list representation
type WeightedAdjList = sequence<sequence<(int,real)>>

fun bellman_ford(adj : WeightedAdjList, source : int) -> option<sequence<real>>:
  n = |adj|
  dist = [∞ for _ in 0...n-1]
  dist[source] = 0
  for k in 1...n:
    for u in 0...n-1:
      for v, w in adj[u]:
        if dist[u] + w < dist[v]:
          if k == n: return NONE  // negative cycle detected!
          dist[v] ← dist[u] + w
  return SOME(dist)
```

Alternative interpretation of Bellman-Ford: Find violations of the triangle inequality and fix them. This is called "relaxing" an edge.

Cost of Bellman-Ford

Theorem (Cost of Bellman-Ford): The cost of (sequential) Bellman-Ford is $O(nm)$ work.

- *The algorithm does n iterations to compute n-hop paths*
- *Each iteration loops over every edge in G: costs $O(m)$*
- *Therefore, the total cost is $O(nm)$*

How about making it **parallel**?

Parallelizing Bellman-Ford

- We can't compute the k-hop paths before we have computed the $(k-1)$-hop paths, so that part is sequential
- Otherwise, for each vertex v, its k-hop distance is:

$$\mathbf{Dist}(v,k) = \min\left\{\mathbf{Dist}(v,k-1), \min_{u \in N^-(v)}\left(\mathbf{Dist}(u,k-1) + w(u,v)\right)\right\}$$

- This can be computed with a **reduce**!
- Each vertex v is independent and can be done in parallel

Note: This requires the **reverse adjacency list** again

Parallel Bellman-Ford

We want to **parallelize** the computation of the k-hop paths as a function of the $(k-1)$-hop paths

```
for v in 0...n-1:
  dist[v][k] ← dist[v][k-1]
  for u, w in in_adj[v]:
    dist[v][k] ← min(dist[v][k], dist[u][k-1] + w)
```

```
tabulate(fn (v : int) =>
  min(prev_dist[v],
    reduce(min, ∞, map(fn (u,w) => prev_dist[u] + w, in_adj[v]))), n)
```

Parallel Bellman-Ford

```
type RevAdjList = sequence<sequence<(int,real)>>

fun bellman_ford(in_adj : RevAdjList, s : int) -> option<sequence<real>>:
  n = |in_adj|

  fun hop(prev_dist : sequence<real>, k : int) -> sequence<real>:
     return tabulate ((fn (v : int) =>
       min(prev_dist[v],
           reduce(min, ∞, map(fn (u,w) => prev_dist[u] + w, in_adj[v])))), n))

  init = tabulate(fn v => 0 if v == s else ∞, n)
  dist = fold_left(hop, init, 1...n-1)

  if hop(dist) != dist: return NONE  // Negative cycle!
  return SOME(dist)
```

Cost of Parallel Bellman-Ford

Theorem (Cost of Parallel Bellman-Ford): The cost of parallel Bellman-Ford is $O(nm)$ work and $O(n \log n)$ span.

- *The n hops are computed one by one, so these n steps are all entirely sequential*
- *Each step computes the k-hop distances for each vertex in parallel using **reduce***
- *Reduce costs $O(\log n)$ span*
- *Hence the total span is $O(n \log n)$*

All-Pairs Shortest Paths: Floyd-Warshall

All-Pairs Shortest Paths

Problem (All-Pairs Shortest Path): We want to find, for all $u, v \in V$, a shortest path from u to v

- The simplest way to solve all-pairs shortest paths is…
- Use a single-source shortest path algorithm n times

Theorem (APSP in unweighted graphs): In an unweighted graph, APSP can be solved in $O\big(n(m+n)\big)$ work.

Run BFS n times

All-Pairs Shortest Paths

Problem (All-Pairs Shortest Path): We want to find, for all $u, v \in V$, a shortest path from u to v

- The simplest way to solve all-pairs shortest paths is…
- Use a single-source shortest path algorithm n times

Theorem (APSP in non-negative weighted graphs): In a weighted graph with no negative weights, APSP can be solved in $O(mn \log n)$ work.

Run Dijkstra's algorithm n times

All-Pairs Shortest Paths

Problem (All-Pairs Shortest Path): We want to find, for all $u, v \in V$, a shortest path from u to v

- The simplest way to solve all-pairs shortest paths is…
- Use a single-source shortest path algorithm n times

Theorem (APSP in negative weighted graphs): In a weighted graph with negative weights, APSP can be solve in $O(mn^2)$ work.

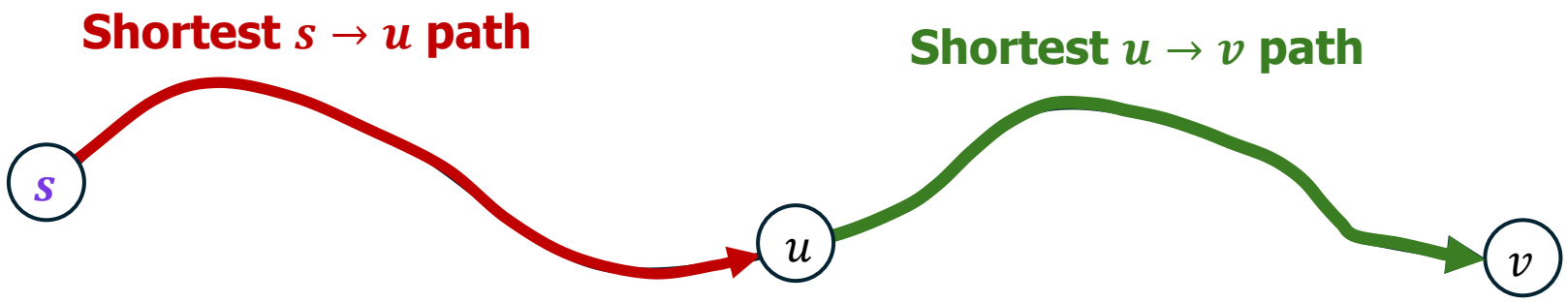
Run Bellman-Ford n times

APSP but Faster

Theorem (Sub-paths property): Any sub-path of a shortest path is itself also a shortest path between its endpoints

- So far, we have only used the sub-paths property to express one path in terms of a smaller path plus **one** additional edge

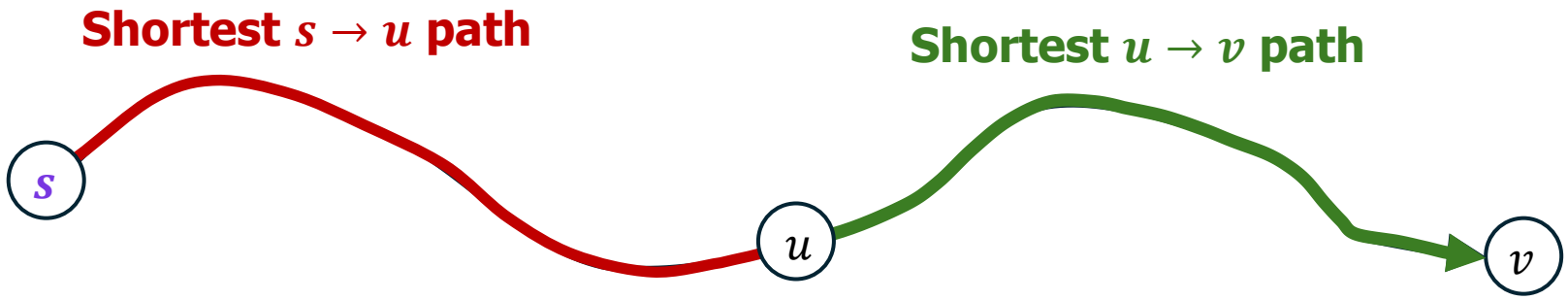
Remember: the subpaths property allows us to break the path **anywhere**



Recurrence for APSP

- We started with the following recurrence based on the sub-paths property: $\delta(s,v) = \min_{u \in N^-(v)} \big(\delta(s,u) + w(u,v)\big)$
- To exploit the sub-paths property more fully, we can split a path at an intermediate vertex, rather than just remove its last edge.

$$\delta(s,v) = \min_{u \in V} \big(\delta(s,u) + \delta(u,t)\big)$$



Recurrence for APSP

$$\delta(s,v) = \min_{u \in V}\big(\delta(s,u) + \delta(u,t)\big)$$

This raw recurrence has the same problem as the earlier one: it is **cyclic** so we can't just evaluate it, or it would produce **infinite recursion**!

- Bellman–Ford made the recurrence acyclic by restricting the **number of edges** a path could use
- We need a similar parameter that **makes both subproblems smaller**

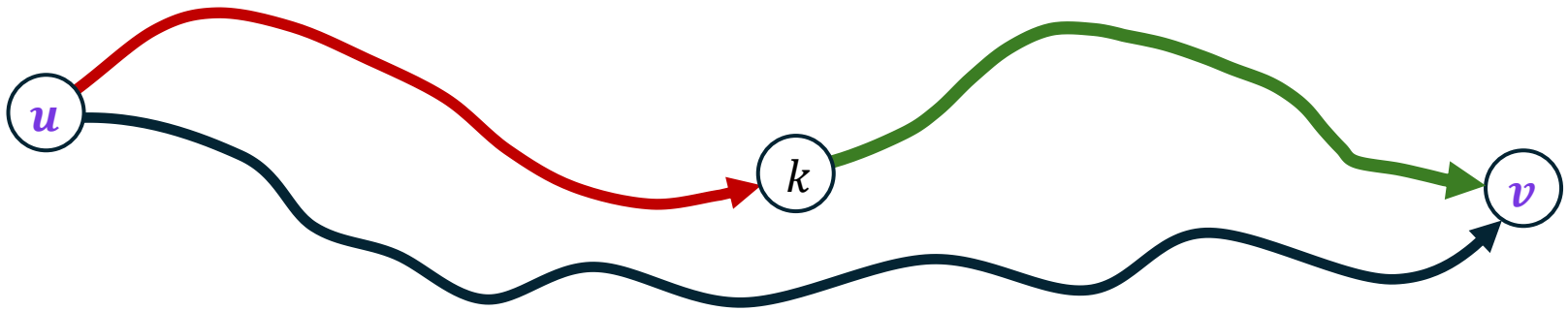
Key Idea: Instead of limiting the number of edges, **limit the set of vertices** that a path is allowed to pass through.

Subproblems for APSP

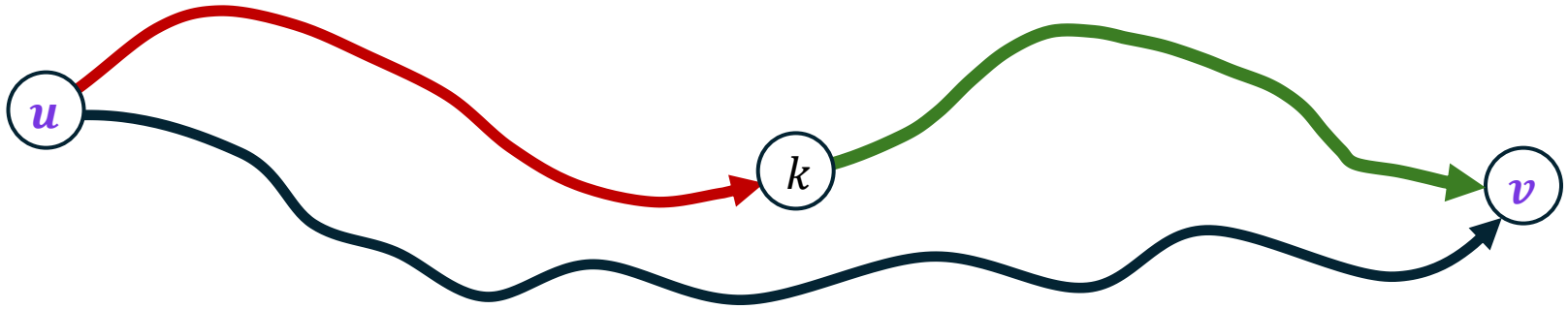
- Define the subproblems

$$\text{Dist}(u, v, k) := \begin{cases} \text{weight of the shortest path from } u \text{ to } v \\ \text{using intermediate vertices } \{1, 2, \ldots, k\} \end{cases}$$

- Each recursive call should be on a **smaller value of k**, i.e., a smaller set of vertices. This makes the recurrence acyclic



Subproblems for APSP



$$\text{Dist}(u, v, k) = \min\left(\begin{cases}\text{Dist}(u, v, k-1) \\ \text{Dist}(u, k, k-1) + \text{Dist}(k, v, k-1)\end{cases}\right)$$

The recurrence is **clever brute force**: Either k is on the shortest path from u to v or it isn't. Try both options and take the best one!

Representation for APSP

- Since the result is $\Theta(n^2)$ size and we want the distance between every pair of vertices, an **adjacency matrix** is a good choice

Notice that the adjacency matrix is **also the base case of our DP**: the shortest-path distances when no intermediate vertices are allowed.

$$\text{Dist}(u, v, 0) = \begin{cases} 0 & \text{if } u = v, \\ w(u, v) & \text{if } (u, v) \in E, \\ \infty & \text{otherwise.} \end{cases}$$

The Floyd-Warshall Algorithm

```
// assume a directed adjacency matrix representation
type WeightedAdjMatrix = sequence<sequence<real>>

fun floyd_warshall(D : WeightedAdjMatrix):
  n = |D|
  // After iteration k, D[u][v] = weight of the shortest
  // u->v path that's allowed to use vertices in the range 0...k
  for k in 0...n-1:
    for u in 0...n-1:
      for v in 0...n-1:
        D[u][v] ← min(D[u][v], D[u][k] + D[k][v]);
  return D
```

Subtle observation: We can update D **in place** because during iteration k, the entire k^{th} row and column remain unchanged!

Parallelizing Floyd-Warshall

We want to **parallelize** the computation of the shortest paths allowing vertex k, using the shortest paths that only allow vertices $0, \ldots, k-1$

```
for u in 0...n-1:
  for v in 0...n-1:
    D[u][v] ← min(D[u][v], D[u][k] + D[k][v])
```

```
tabulate(fn u => tabulate(fn v =>
   min(P[u][v], P[u][k] + P[k][v]),
n), n)
```

Doesn't require a reduce like Bellman-Ford because it's just the min of two values!

Parallelizing Floyd-Warshall

- The n main steps of the outer loop are still entirely sequential, each vertex k must be considered one at a time
- However, every pair of vertices u, v can be evaluated in parallel

```
fun floyd_warshall(D : WeightedAdjMatrix):
  n = |D|

  fun next_k(P : WeightedAdjMatrix, k : int) -> WeightedAdjMatrix:
    return tabulate(fn u => tabulate(fn v =>
              min(P[u][v], P[u][k] + P[k][v]),
            n), n)

  return fold_left(next_k, D, 0...n-1)
```

Cost of Floyd-Warshall

Theorem (Cost of Floyd-Warshall): (Parallel) Floyd-Warshall costs $O(n^3)$ work and $O(n)$ span

- *Folding over the n values of k is sequential, but for each value of k each of the u-v pairs is computed in parallel*
- *For each u-v pair the algorithm computes a min of two values which costs constant work*
- *Hence the total cost is $O(n^3)$ work and $O(n)$ span*

Summary

- The **Bellman-Ford algorithm** solves single-source shortest paths on graphs with negative weights in $O(mn)$ work (and $O(n \log n)$ span for the parallel version).
- Bellman-Ford is based on the idea of **k-hop shortest paths**
- The **all-pairs shortest paths** problem can be solved on weighted graphs (including negative weights) in $O(n^3)$ work and $O(n)$ span (The **Floyd-Warshall Algorithm**).
- In graphs with **negative-weight cycles**, shortest paths may be undefined. We can detect this with Bellman-Ford.