Parallel And Sequential Data Structures and Algorithms

Shortest Paths II: Bellman-Ford and Floyd-Warshall

Announcements

- **TA applications** are still open!
  - https://forms.gle/2ijyhz7yzxCg2zbS6
- **Midterm Two grades** will be out tonight

Learning Objectives

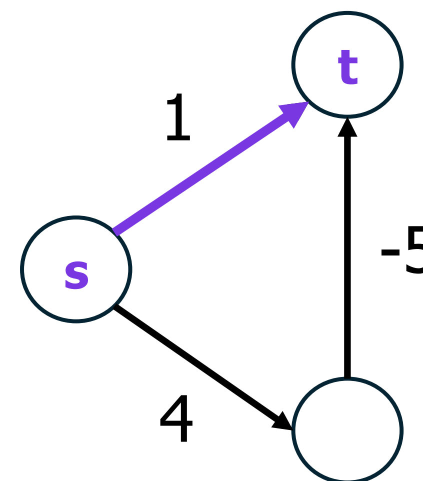
- See algorithms that solve for shortest paths on graphs with negative weights (**Bellman-Ford**)
- See how to detect the presence of **negative-weight cycles**, which may make shortest paths undefined
- Solve the all-pairs shortest paths problem (on graphs with negative weights), with the **Floyd-Warshall** Algorithm

Shortest Paths with Negative Weights

Negative Weights and Dijkstra

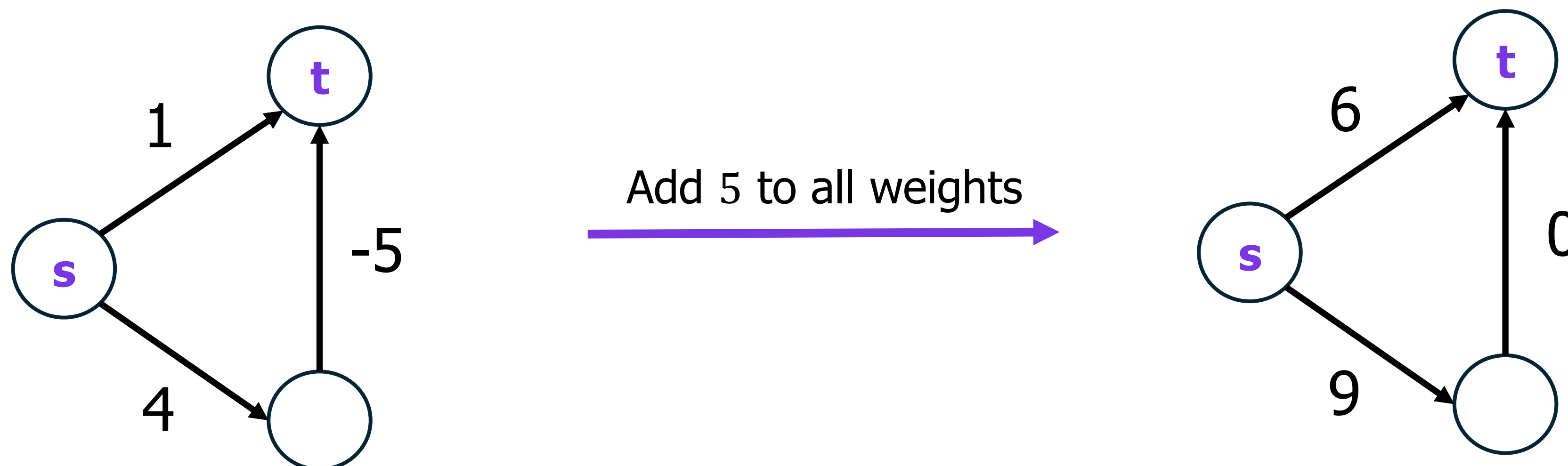
What will Dijkstra's Algorithm do on this graph with negative weights?

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)

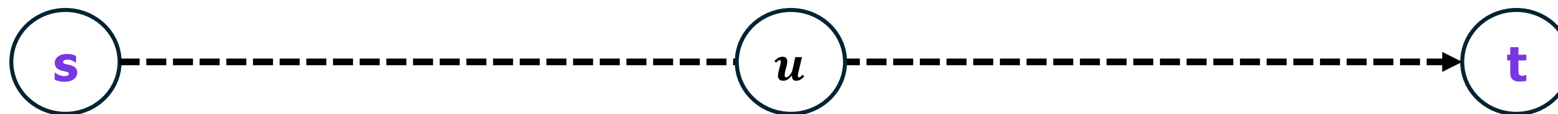
- A shortest-path algorithm for non-negative weights will always fine a valid answer (provided that s can reach t)
- For negative weights, we have a choice:
  - Detect a negative-weight cycle and report invalid paths
  - Just silently return bogus answers
- We will discuss how to detect negative cycles

k-hop Distances and Bellman-Ford

Shortest Paths with Negatives

- The issue with Dijkstra's algorithm in the presence of negative weights, is that **adding edges can make a path have a lower total weight**
- The following property is still true however:

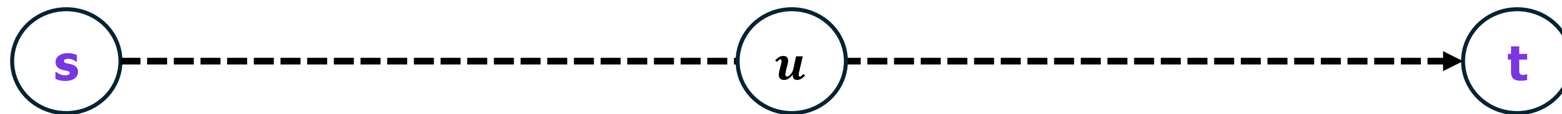
Theorem (Sub-paths): A sub-path of a shortest path is itself also a shortest path, even in the presence of negative weights



Sub-Paths

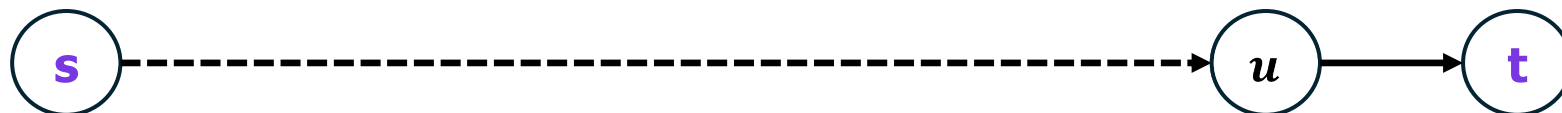
- Sub-paths property is true no matter where we split the path.
- That is, if u is on a shortest path from s to t, then

$$\delta(s,t) = \delta(s,u) + \delta(u,t)$$



- Also, if v is the second-last vertex on a shortest path from s to t

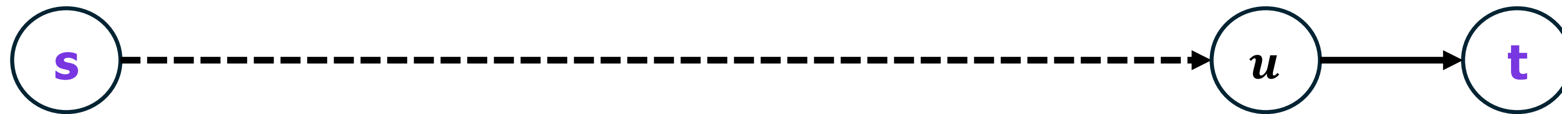
$$\delta(s,t) = \delta(s,u) + w(u,t)$$



k-hop Shortest Paths

- If v is the second-last vertex on a shortest path from s to t

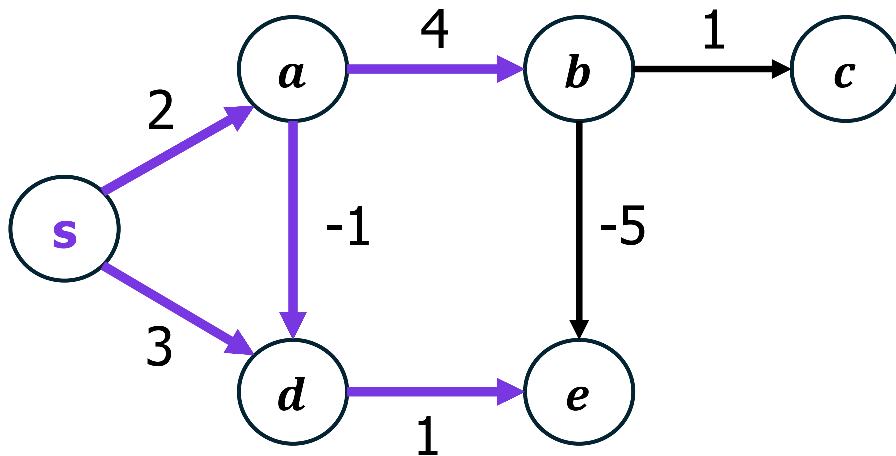
$$\delta(s,t) = \delta(s,u) + w(u,t)$$



The sub-paths property implies something important: the shortest path that contains k edges is composed of **some other shortest path** that contains $k-1$ edges **plus one more edge**.

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.

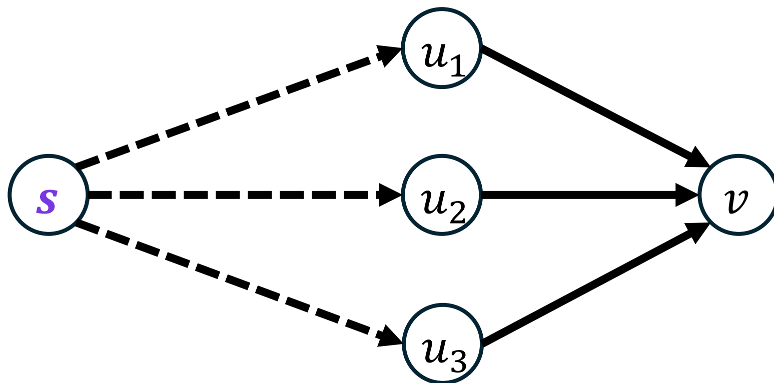
Corollary (Enough Hops): In a graph with no negative-weight cycles, the shortest path distances are the $(n - 1)$-hop distances.

Dynamic Programming for k-hop

- Since a k-hop shortest path is just a $(k-1)$-hop shortest path and one additional edge, this is a perfect problem for DP

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

- The k-hop distance should consider all possible vertices u that end a $(k-1)$-hop path, and add the edge (u, v)



Observe: A k-hop path to v is a $(k-1)$-hop path to u_1, u_2, or u_3 plus one more edge.

Dynamic Programming for k-hop

- Since a k-hop shortest path is just a $(k-1)$-hop shortest path and one additional edge, this is a perfect problem for DP

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

- The k-hop distance should consider all possible vertices u that end a $(k-1)$-hop path, and add the edge (u,v)

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

k-hop Pseudocode

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

fun khopPaths(in_adj : rev_adjacency_list, 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 vertex v such that the n-hop shortest path from s to v is shorter than the corresponding $(n-1)$-hop shortest path from s to v if and only if there exists a negative weight cycle reachable from s.

- So, compute the n-hop shortest paths and see if they differ from the $(n-1)$-hop shortest paths!

The Bellman-Ford Algorithm

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

fun bellman_ford(adj : adjacency_list, 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   // neg
          dist[v] ← dist[u] + w
  return SOME(dist)
```

Alterative 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* $O(nm)$

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:

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

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

Parallel Bellman-Ford

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

fun bellman_ford(in_adj : rev_adjacency_list, s : int) -> option<sequence<real>>:
  n = |in_adj|
  fn hop(dist : sequence<T>, k : int) =>
     map((fn (v : int) =>
      min(dist[v],
          reduce(min, ∞,
                 map(fn (u, w) => dist[u] + w, in_adj[v]))
          ), 0...n-1))
  init = parallel [0 if u == s else ∞ for u in 0...n-1]
  dist = fold_left(hop, init, 1...n)
  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 $(k+1)$-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, we can solve the APSP problem in $O\big(n(m+n)\big)$ cost.

- *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, we can solve the APSP problem in $O(mn \log n)$ cost.

- *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, we can solve the APSP problem in $O(mn^2)$ cost.

- *Run Bellman-Ford n times*

APSP but Faster

- Recall the sub-paths property:

Theorem (Sub-paths): A sub-path of a shortest path is itself also a shortest path, even in the presence of negative weights

- Shortest paths have a ***lot*** of overlap, and we want to compute all of them now
- Dynamic programming sounds appealing yet again

DP for APSP?

- We could take inspiration from Bellman-Ford and do something similar: build paths one extra edge at a time!

$$\text{Dist}(u, v, \ell) := \begin{cases} \text{the weight of the shortest path between} \\ u \text{ and } v \text{ consisting of } \ell \text{ edges.} \end{cases}$$

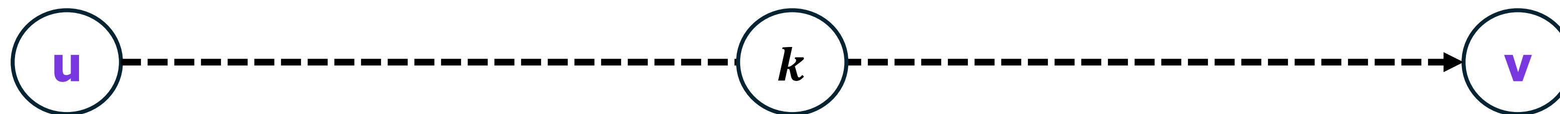
- This works but… there's a problem

This literally is **Bellman-Ford again**, just n times.

Better DP for APSP

- We want to take more advantage of subpaths than just adding a single edge each time
- Remember that sub-paths can be split anywhere, i.e., for any vertex k on a shortest path from s to t

$$\delta(u, v) = \delta(u, k) + \delta(k, v)$$



Dynamic Programming Idea: Try including k on every path, and take the shorter of including it or not including it

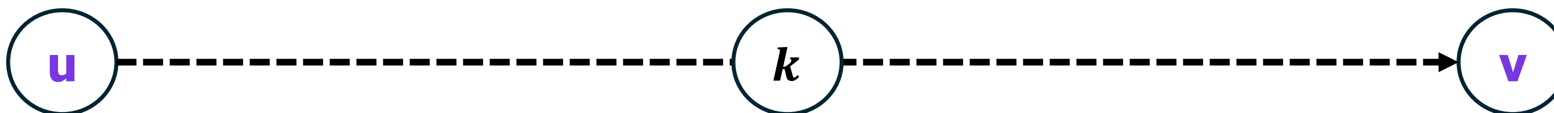
Correct DP for APSP

- Define the subproblems

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

- To solve a subproblem $\text{Dist}(u,v,k)$ we just need to compare whether adding k somewhere along the u-v path improves it:

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



The Floyd-Warshall Algorithm

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

fun floyd_warshall(D : adj_matrix):
  // After each iteration, D[u][v] = weight of the shortest
  // u->v path that's allowed to use vertices in the set 1..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
```

- Note that we employ an **adjacency matrix** for this algorithm

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 : adj_matrix):
  fn next_k(P : adj_matrix, k : int) =>
    tabulate(fn u => tabulate(fn v =>
      min(P[u][v], P[u][k] + P[k][v]),
    0...n-1), 0...n-1)
  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.