Exam 1 Review

CSE 101

Winter 2023

Exam Details

- In class
- Randomized assigned seats
- You may use 6 one-sided pages of notes
- No textbook or electronic aids
- No need to provide proofs unless asked for
- 3 Questions in 45 minutes
  - 1st straightforward implementation of algorithm
  - 2nd requires some thought
  - 3rd can be quite tricky

This Review

- Brief outline of topics that might show up on the exam
- To see anything in more depth use other review options.

Other Review Options

- Lecture podcasts / slides
- Textbook
- OH questions
- Old exams from problem archive

Exam Topics

- Chapter 3
  - Graph basics
  - Explore/DFS
  - Connected components
  - Pre/Post orderings
  - DAGs
  - Topological sort
  - Strongly connected components
- Chapter 4
  - Shortest path definitions
  - BFS
  - Dijkstra
  - Priority queues
  - Bellman-Ford
  - Negative weight cycles
  - Shortest paths in DAGs

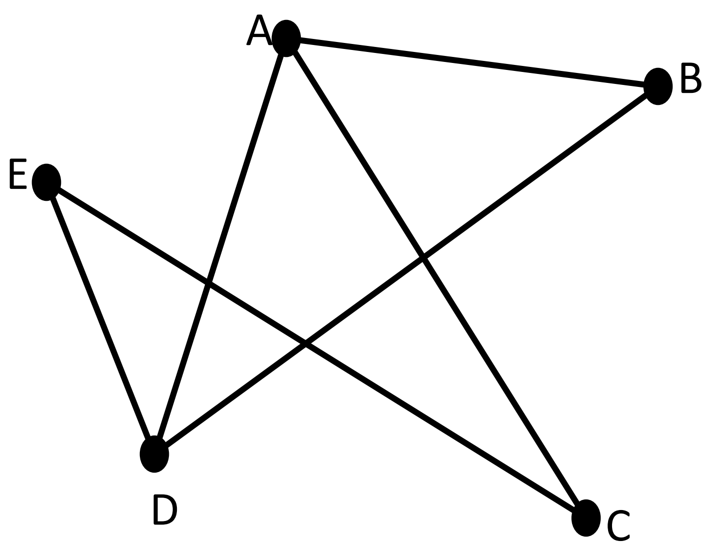
Graph Definition

Definition: A *graph* G = (V,E) consists of two things:

- A collection V of *vertices*, or objects to be connected.
- A collection E of *edges*, each of which connects a pair of vertices.

Drawing Graphs

- Draw vertices as points
- Draw edges as line segments or curves connecting those points



V = {A,B,C,D,E}
E = {AB,AC,AD,
BD,CE,DE}

Explore

```
explore(v)
   v.visited ← true
   For each edge (v,w)
      If not w.visited
         explore(w)
         w.prev ← v
```

Result

Theorem: If all vertices start unvisited, `explore(v)` marks as visited exactly the vertices reachable from v.

Depth First Search

`explore` only finds the part of the graph reachable from a single vertex. If you want to discover the entire graph, you may need to run it multiple times.

```
DepthFirstSearh(G)
  Mark all v ∈ G as unvisited
  For v ∈ G
    If not v.visited, explore(v)
```

Runtime of DFS

```
explore(v)
  v.visited ← true
  For each edge (v,w)
     If not w.visited
          explore(w)
```

Run once per vertex

O(|V|) total

Run once per neighboring vertex

O(|E|) total

```
DFS(G)
  Mark all v ∈ G as unvisited
  For v ∈ G
    If not v.visited, explore(v)
```

O(|V|)

Final runtime:
O(|V|+|E|)

Connected Components

Theorem: The vertices of a graph G can be partitioned into *connected components* so that v is reachable from w if and only if they are in the same connected component.

Computing CCs with DFS

```
ConnectedComponents(G)
  CCNum ← 0
  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      CCNum++
      explore(v)
```

```
explore(v)
  v.visited ← true
  v.CC ← CCNum
  For each edge (v,w)
    If not w.visited
      explore(w)
```

Runtime $O(|V|+|E|)$.

Pre- and Post- Orders

- Keep track of what DFS does & in what order.
- Have a “clock” and note time whenever:
  - Algorithm visits a new vertex for the first time.
  - Algorithm finishes processing a vertex.
- Record values as `v.pre` and `v.post`.

Computing Pre- & Post- Orders

```
PreAndPost(G)
  clock ← 1
  For v ∈ G
    v.visited ← false
  For v ∈ G
    If not v.visited
      explore(v)
```

```
explore(v)
  v.visited ← true
  v.pre ← clock
  clock++
  For each edge (v,w)
    If not w.visited
      explore(w)
  v.post ← clock
  clock++
```

Runtime $O(|V|+|E|)$.

What do these orders tell us?

Prop: For vertices v, w consider intervals [`v.pre`, `v.post`] and [`w.pre`, `w.post`]. These intervals:

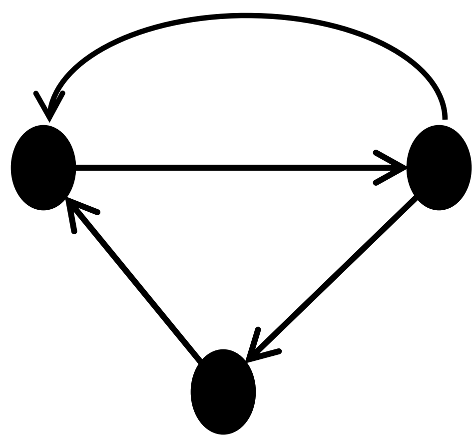
1. Contain each other if v is an ancestor/descendant of w in the DFS tree.
2. Are disjoint if v and w are cousins in the DFS tree.
3. Never interleave (`v.pre` < `w.pre` < `v.post` < `w.post`)

Directed Graphs

Often an edge makes sense both ways, but sometimes streets are one directional.

Definition: A directed graph is a graph where each edge has a direction. Goes *from* v *to* w.

Draw edges with arrows to denote direction.

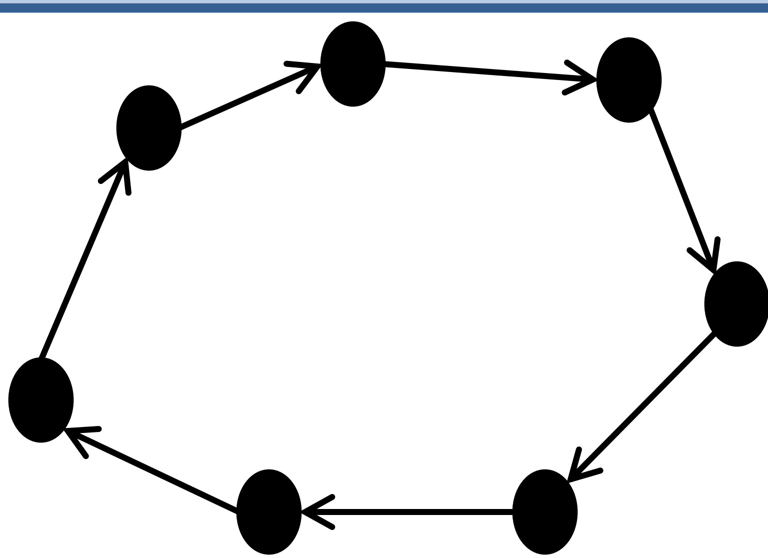


Dependency Graphs

Definition: A topological ordering of a directed graph is an ordering of the vertices so that for each edge (v,w), v comes before w in the ordering.

Cycles

<u>Definition:</u> A <u>cycle</u> in a directed graph is a sequence of vertices v_1, v_2, v_3,…,v_n so that there are edges (v_1,v_2), (v_2,v_3),…,(v_{n-1},v_n),(v_n,v_1)



DAGs

Definition: A Directed Acyclic Graph (DAG) is a directed graph which contains no cycles.

Theorem: A directed graph G has a topological ordering if and only if it is a DAG.

Topological Sort

```
TopologicalSort(G)
  Run DFS(G) w/ pre/post numbers
  Return the vertices in reverse
 postorder
```

Runtime: $O(|V|+|E|)$.

Correctness

<u>Proposition:</u> If G is a DAG with an edge v → w then `w.post < v.post`.

Strongly Connected Components

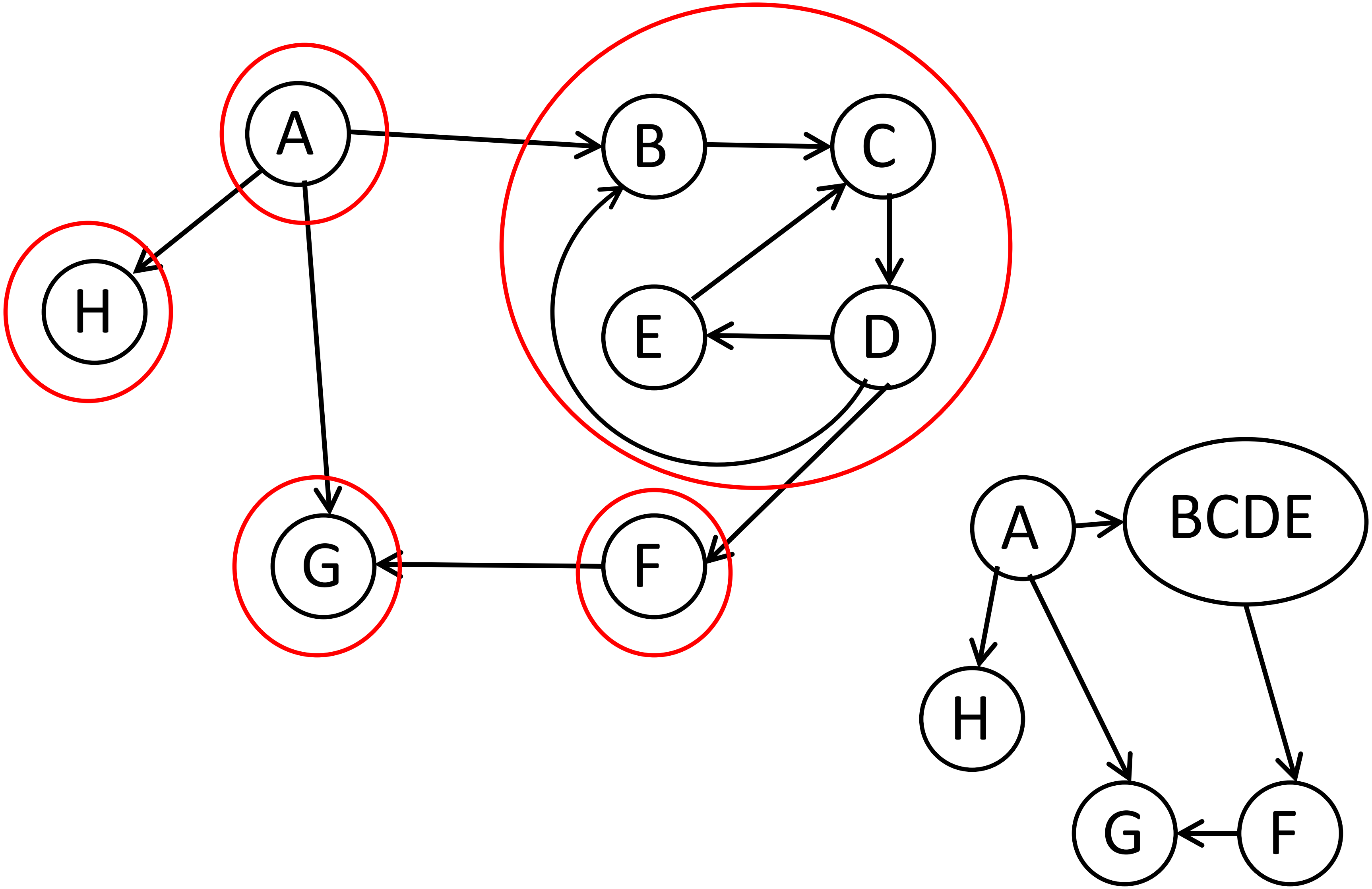
Definition: In a directed graph G, two vertices v and w are in the same Strongly Connected Component (SCC) if v is reachable from w *and* w is reachable from v.

Lemma: You can actually partition the vertices into components in this way.

Metagraph

<u>Definition:</u> The <u>metagraph</u> of a directed graph G is a graph whose vertices are the SCCs of G, where there is an edge between C_1 and C_2 if and only if G has an edge between some vertex of C_1 and some vertex of C_2.

Example



Result

<u>Theorem:</u> The metagraph is any directed graph is a DAG.

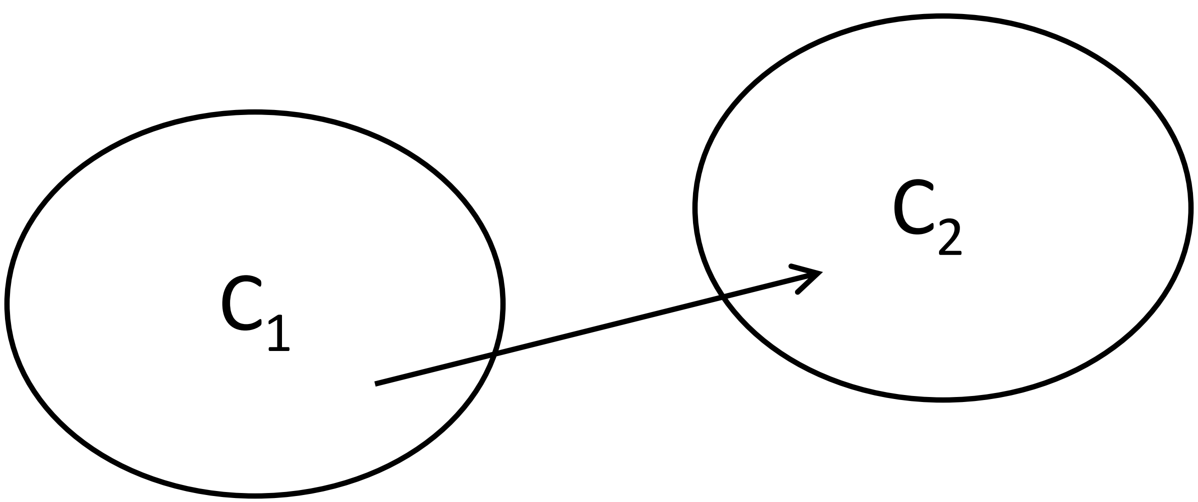
<u>Problem:</u> Given a directed graph G compute the SCCs of G and its metagraph.

Observation

If v in sink SCC, `explore(v)` finds *exactly* v's component.

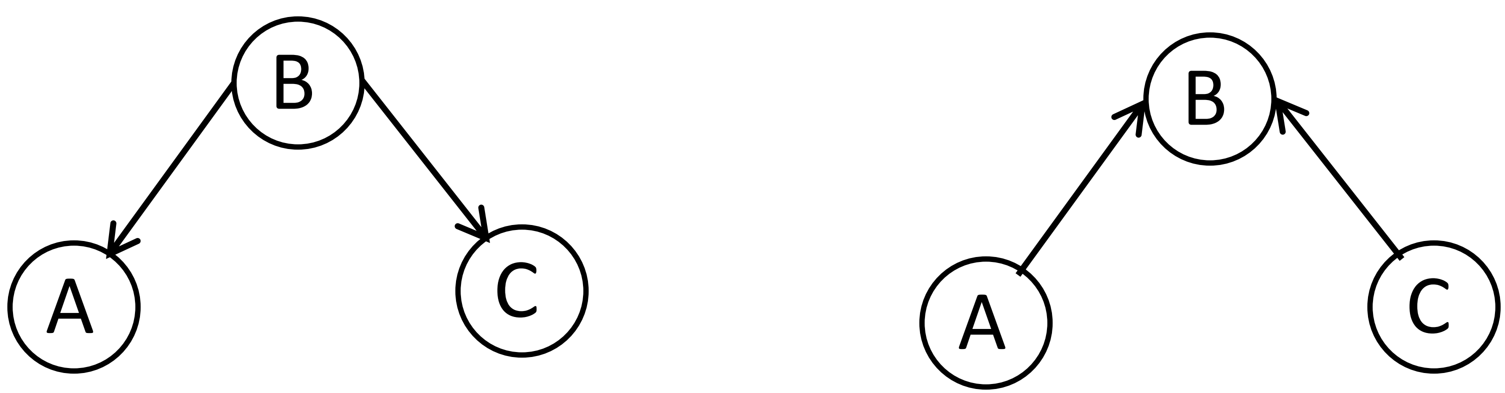
Result

Proposition: Let C_1 and C_2 be SCCs of G with an edge from C_1 to C_2. If we run `DFS` on G, the largest postorder number of any vertex in C_1 will be larger than the largest postorder number in C_2.



Reverse Graph

<u>Definition:</u> Given a directed graph G, the <u>reverse graph</u> of G (denoted G^R) is obtained by reversing the directions of all of the edges of G.



SCC Algorithm

```
SCCs(G)
  Run DFS(G^R) record postorders
  Mark all vertices unvisited
  For v ∈ V in reverse postorder
    If not v.visited
      explore(v) mark component
```

Just 2 DFSs! Runtime $O(|V|+|E|)$.

Goal

<u>Problem:</u> Given a graph G with two vertices s and t in the same connected component, find the *best* path from s to t.

What do we mean by best?

- Least expensive
- Best scenery
- Shortest
- For now: fewest edges

Observation

If there is a length ≤d s-v path, then there is some w adjacent to v with a length ≤(d-1) s-w path.

Algorithm Idea

For each d create a list of all vertices at distance d from s.

- For d=0, this list is just {s}.
- For larger d, we want all new vertices adjacent to vertices at distance d-1.

Breadth-First Search

```
BFS(G,s)
  For v ∈ V, dist(v) ← ∞
  Initialize Queue Q
  Q.enqueue(s)
  dist(s) ← 0
  While(Q nonempty)
    u ← front(Q)
    For (u,v) ∈ E
      If dist(v) = ∞
        dist(v) ← dist(u)+1
        Q.enqueue(v)
        v.prev ← u
```

O(|V|)

O(|V|) iterations

O(|E|) total iterations

Total runtime: O(|V|+|E|)

Edge Lengths

The number of edges in a path is not always the right measure of distance. Sometimes, taking several shorter steps is preferable to taking a few longer ones.

We assign each edge (u,v) a non-negative <u>length</u> $\ell(u,v)$. The length of a path is the sum of the lengths of its edges.

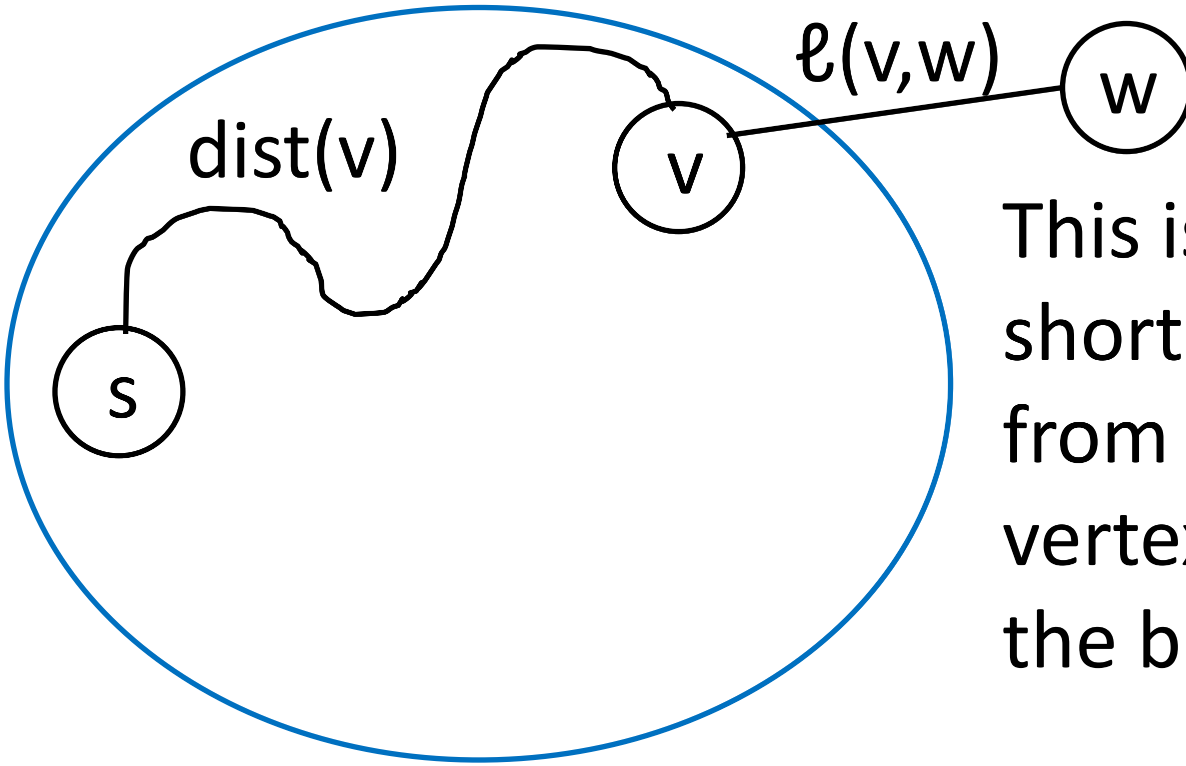
Problem: Shortest Paths

Problem: Given a Graph G with vertices s and t and a length function ℓ, find the shortest path from s to t.

Algorithm

```
Distances(G,s,ℓ)
  dist(s) ← 0
  While(not all distances found)
    Find minimum over (v,w) ∈ E
      with v discovered w not
      of dist(v)+ℓ(v,w)
    dist(w) ← dist(v)+ℓ(v,w)
    prev(w) ← v
```

Why does this work?



This is the shortest path from s to *any* vertex outside the bubble.

Correctly Assigned Distances

Runtime

- This is too slow.
- **Problem:** Every iteration we have to check every edge.
- **Idea:** Most of the comparison doesn't change much iteration to iteration. Use to save time.
- **Use data structure:** In particular a priority queue.

Priority Queue

A <u>Priority Queue</u> is a datastructure that stores elements sorted by a <u>key</u> value.

<u>Operations:</u>

- Insert – adds a new element to the PQ.
- DecreaseKey – Changes the key of an element of the PQ to a specified *smaller* value.
- DeleteMin – Finds the element with the smallest key and removes it from the PQ.

Dijkstra's Algorithm

```
Dijkstra(G,s,ℓ)
  Initialize Priority Queue Q
  For v ∈ V
    dist(v) ← ∞
    Q.Insert(v)
  dist(s) ← 0
  While(Q not empty)
    v ← Q.DeleteMin()
    For (v,w) ∈ E
      If dist(v)+ℓ(v,w) < dist(w)
        dist(w) ← dist(v)+ℓ(v,w)
        Q.DecreaseKey(w)
```

O(|V|) times

O(|V|) times

O(|E|) times

Runtime:
O(|V|) Inserts +
O(|V|) DelMins +
O(|E|) DecKeys

Summary of Priority Queues

	Insert/DecreaseKey	DeleteMin	Dijkstra
List	$O(1)$	$O(n)$	$O(\|V\|^2 + \|E\|)$
Binary Heap	$O(\log(n))$	$O(\log(n))$	$O(\log \|V\|(\|V\| + \|E\|))$
d-ary Heap	$O\left(\frac{\log(n)}{\log(d)}\right)$	$O\left(\frac{d\log(n)}{\log(d)}\right)$	$O\left(\frac{\log \|V\|}{\log(d)}(d\|V\| + \|E\|)\right)$
Fibonacci Heap	$O(1)^*$	$O(\log(n))^*$	$\boxed{O(\|V\| \log \|V\| + \|E\|)}$

Negative Edge Weights

- So far we have talked about the case of non-negative edge weights.
  - The usual case (distance & time usually cannot be negative).
  - However, if “lengths” represent other kinds of costs, sometimes they can be negative.
- Problem statement same. Find path with smallest sum of edge weights.

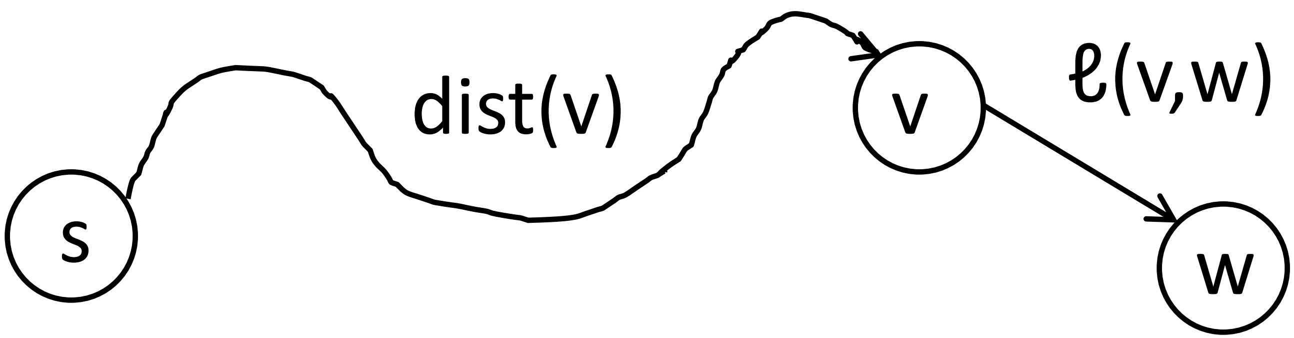
Negative Weight Cycles

Definition: A negative weight cycle is a cycle where the total weight of edges is negative.

- If G has a negative weight cycle, then there are probably no shortest paths.
  - Go around the cycle over and over.
- **Note:** For undirected G, a single negative weight edge gives a negative weight cycle by going back and forth on it.

Fundamental Shortest Paths Formula

$$\text{For } w \neq s,$$
$$\operatorname{dist}(w) = \min_{(v,w)\in E} \operatorname{dist}(v) + \ell(v,w).$$

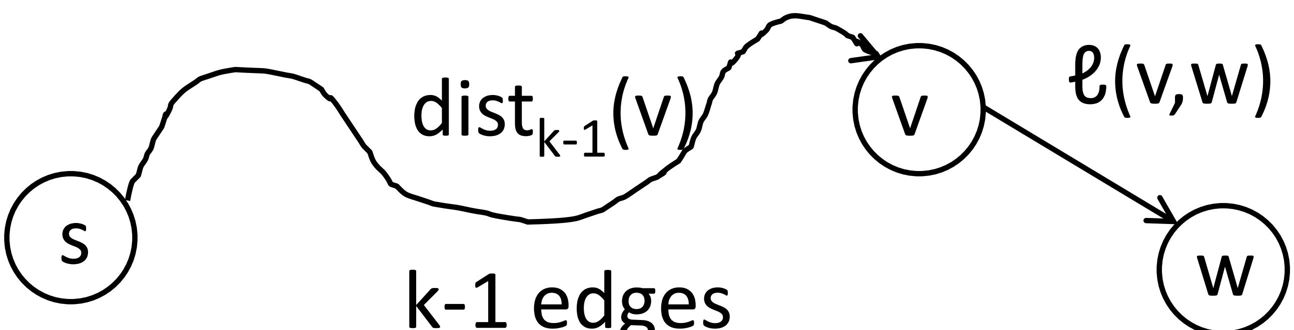


- System of equations to solve for distances.
- When $\ell \geq 0$, Dijsktra gives an order to solve in.
- With $\ell < 0$, might be no solution.

Algorithm Idea

Instead of finding shortest paths (which may not exist), find shortest paths of length at most k.

$$\text{For } w \neq s,$$
$$\text{dist}_k(w) = \min_{(v,w)\in E} \text{dist}_{k-1}(v) + \ell(v,w).$$



Bellman-Ford

```
Bellman-Ford(G,s,ℓ)
  dist_0(v) ← ∞ for all v
    //cant reach
  dist_0(s) ← 0
  For k = 1 to n
    For w ∈ V
      dist_k(w)←min(dist_{k-1}(v)+ℓ(v,w))
    dist_k(s) ← min(dist_k(s),0)
      // s has the trivial path
```

What value of k do we use?

O(|E|)

Analysis

Proposition: If $n \geq |V|-1$ and if G has no negative weight cycles, then for all v,

$$\text{dist}(v) = \text{dist}_n(v).$$

- If there is a negative weight cycle, there probably is no shortest path.
- If not, we only need to run our algorithm for $|V|$ rounds, for a final runtime $O(|V||E|)$.

Detecting Negative Cycles

If there are no negative weight cycles, Bellman-Ford computes shortest paths (and they might not exist otherwise).

How do we know whether or not there are any?

Cycle Detection

Proposition: For any $n \geq |V| - 1$, there are no negative weight cycles reachable from s if and only if for every $v \in V$

$$dist_n(v) = dist_{n+1}(v)$$

- Detect by running one more round of Bellman-Ford.
- Need to see if *any* v's distance changes.

Shortest Paths in DAGs

Runtime
O(|V|+|E|)

```
ShortestPathsInDAGs(G,s,ℓ)
  TopologicalSort(G)
  For w ∈ V in topological order
    If w = s, dist(w) ← 0
    Else
      dist(w)←min(dist(v)+ℓ(v,w))
\\ dist(v) for all upstream v
   already computed
```

O(|V|+|E|)

O(|E|) total

O(|V|) total

Shortest Path Algorithms Summary

Unit Weights: `Breadth First Search` $O(|V|+|E|)$

Non-negative Weights: `Dijkstra` $O(|V|\log|V|+|E|)$

Arbitrary Weights: `Bellman-Ford` $O(|V||E|)$

Arbitrary Weights, graph is a DAG: `Shortest-Paths-In-DAGs` $O(|V|+|E|)$