Shortest paths; Dijkstra's algorithm

CS 2860: Algorithms and Complexity

Magnus Wahlström and Gregory Gutin

October 15, 2017

Shortest path problems

Shortest path problems

- Ubiquitous question – What is the best/fastest/cheapest way to get from A to B?
- Natural graph interpretation
- Variants:
  - Directed or undirected?
  - Weights: No weights, positive weights, positive and negative weights?
  - Questions: From A to B? From A to everywhere? All-to-all?

Shortest path problems

- Ubiquitous question – What is the best/fastest/cheapest way to get from A to B?
- Natural graph interpretation
- Variants:
  - Directed or undirected?
  - Weights: No weights, positive weights, positive and negative weights?
  - Questions: From A to B? From A to everywhere? All-to-all?

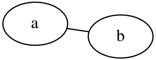
Shortest path problems

- Ubiquitous question – What is the best/fastest/cheapest way to get from A to B?
- Natural graph interpretation
- Variants:
  - Directed or undirected?
  - Weights: No weights, positive weights, positive and negative weights?
  - Questions: From A to B? From A to everywhere? All-to-all?

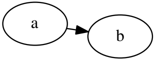
Shortest path problems

- Ubiquitous question – What is the best/fastest/cheapest way to get from A to B?
- Natural graph interpretation
- Variants:
  - Directed or undirected?
  - Weights: No weights, positive weights, positive and negative weights?
  - Questions: From A to B? From A to everywhere? All-to-all?

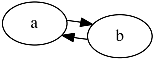
Distinction 1: Directed, undirected?



Undirected



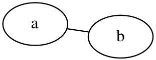
Directed



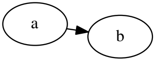
Bidirectional

- Directed edge (arc): $u \to v$
  - Can be followed in the given direction only
- Undirected edge: uv, $\{u, v\}$
  - Can be followed in either direction
- Common reduction:
  - Replace edge uv by two arcs $u \to v$, $v \to u$ – bidirectional arcs
  - No shortest path will want to use both arcs uv and vu
- Our algorithms assume directed

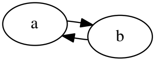
Distinction 1: Directed, undirected?



Undirected



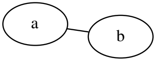
Directed



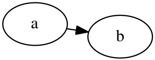
Bidirectional

- Directed edge (arc): $u \to v$
  - Can be followed in the given direction only
- Undirected edge: uv, $\{u, v\}$
  - Can be followed in either direction
- Common reduction:
  - Replace edge uv by two arcs $u \to v$, $v \to u$ – bidirectional arcs
  - No shortest path will want to use both arcs uv and vu
- Our algorithms assume directed

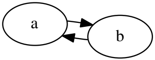
Distinction 1: Directed, undirected?



Undirected



Directed



Bidirectional

- Directed edge (arc): $u \rightarrow v$
  - Can be followed in the given direction only
- Undirected edge: uv, $\{u, v\}$
  - Can be followed in either direction
- Common reduction:
  - Replace edge uv by two arcs $u \rightarrow v$, $v \rightarrow u$ – bidirectional arcs
  - No shortest path will want to use both arcs uv and vu
- Our algorithms assume directed

Distinction 2: Weights type

- Weights – abstract cost notion
  - Roads: Distance, time, cost, . . .
  - Abstract graphs: e.g., time offset (see later)
- Unit weights – all edges are equal
  - We care only about the number of hops in our paths
- Non-negative weights – natural notion
  - Example: Roads long or short, but not (e.g.) minus five meters
- Arbitrary weights – positive or negative
  - Sounds odd – but useful in abstract problems
  - Example (will see): Currency arbitrage

Distinction 2: Weights type

- Weights – abstract cost notion
  - Roads: Distance, time, cost, . . .
  - Abstract graphs: e.g., time offset (see later)
- Unit weights – all edges are equal
  - We care only about the number of hops in our paths
- Non-negative weights – natural notion
  - Example: Roads long or short, but not (e.g.) minus five meters
- Arbitrary weights – positive or negative
  - Sounds odd – but useful in abstract problems
  - Example (will see): Currency arbitrage

Distinction 2: Weights type

- Weights – abstract cost notion
  - Roads: Distance, time, cost, ...
  - Abstract graphs: e.g., time offset (see later)
- Unit weights – all edges are equal
  - We care only about the number of hops in our paths
- Non-negative weights – natural notion
  - Example: Roads long or short, but not (e.g.) minus five meters
- Arbitrary weights – positive or negative
  - Sounds odd – but useful in abstract problems
  - Example (will see): Currency arbitrage

Distinction 2: Weights type

- Weights – abstract cost notion
  - Roads: Distance, time, cost, ...
  - Abstract graphs: e.g., time offset (see later)
- Unit weights – all edges are equal
  - We care only about the number of hops in our paths
- Non-negative weights – natural notion
  - Example: Roads long or short, but not (e.g.) minus five meters
- Arbitrary weights – positive or negative
  - Sounds odd – but useful in abstract problems
  - Example (will see): Currency arbitrage

Distinction 2: Weights type

- Weights – abstract cost notion
  - Roads: Distance, time, cost, ...
  - Abstract graphs: e.g., time offset (see later)
- Unit weights – all edges are equal
  - We care only about the number of hops in our paths
- Non-negative weights – natural notion
  - Example: Roads long or short, but not (e.g.) minus five meters
- Arbitrary weights – positive or negative
  - Sounds odd – but useful in abstract problems
  - Example (will see): Currency arbitrage

Distinction 3: Problem type

- One-to-one – From London to Novosibirsk
  - Usually no faster solution than single-source
  - (But see A* heuristic)
- Single-source: From London to anywhere
  - Focus today
- All-pairs shortest paths
- Dijkstra's algorithm
  - Directed graph, non-negative weights
  - Single source, all destinations

Distinction 3: Problem type

- One-to-one – From London to Novosibirsk
  - Usually no faster solution than single-source
  - (But see A* heuristic)
- Single-source: From London to anywhere
  - Focus today
- All-pairs shortest paths
- Dijkstra's algorithm
  - Directed graph, non-negative weights
  - Single source, all destinations

Distinction 3: Problem type

- One-to-one – From London to Novosibirsk
  - Usually no faster solution than single-source
  - (But see A* heuristic)
- Single-source: From London to anywhere
  - Focus today
- All-pairs shortest paths
- Dijkstra's algorithm
  - Directed graph, non-negative weights
  - Single source, all destinations

Distinction 3: Problem type

- One-to-one – From London to Novosibirsk
  - Usually no faster solution than single-source
  - (But see A* heuristic)
- Single-source: From London to anywhere
  - Focus today
- All-pairs shortest paths
- Dijkstra's algorithm
  - Directed graph, non-negative weights
  - Single source, all destinations

Distinction 3: Problem type

- One-to-one – From London to Novosibirsk
  - Usually no faster solution than single-source
  - (But see A* heuristic)
- Single-source: From London to anywhere
  - Focus today
- All-pairs shortest paths
- Dijkstra's algorithm
  - Directed graph, non-negative weights
  - Single source, all destinations

Single-source shortest paths

Structure of shortest paths

Shortest paths tree

For any graph G, with a source vertex s and non-negative edge lengths, shortest paths from s to all other vertices can be captured via a rooted spanning out-tree (branching).

Observations:

1. There can be several shortest paths from s to v – but we only need one.
2. Any prefix of a shortest path is a shortest paths
   - If the shortest paths from s to v ends in the arc uv, then the sub-path to u is a shortest path from s to u
3. Don't need to consider cycles/loops (shortcut)

Structure of shortest paths

Shortest paths tree

For any graph G, with a source vertex s and non-negative edge lengths, shortest paths from s to all other vertices can be captured via a rooted spanning out-tree (branching).

Observations:

1. There can be several shortest paths from s to v – but we only need one.
2. Any prefix of a shortest path is a shortest paths
   - If the shortest paths from s to v ends in the arc uv, then the sub-path to u is a shortest path from s to u
3. Don't need to consider cycles/loops (shortcut)

Structure of shortest paths

Shortest paths tree

For any graph G, with a source vertex s and non-negative edge lengths, shortest paths from s to all other vertices can be captured via a rooted spanning out-tree (branching).

Observations:

1. There can be several shortest paths from s to v – but we only need one.
2. Any prefix of a shortest path is a shortest paths
   - If the shortest paths from s to v ends in the arc uv, then the sub-path to u is a shortest path from s to u
3. Don't need to consider cycles/loops (shortcut)

Structure of shortest paths

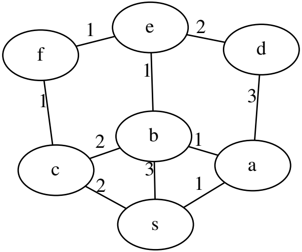
Shortest paths tree

For any graph G, with a source vertex s and non-negative edge lengths, shortest paths from s to all other vertices can be captured via a rooted spanning out-tree (branching).

Observations:

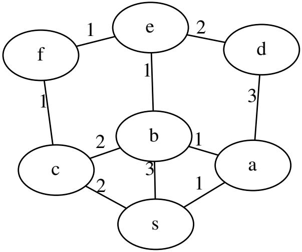
1. There can be several shortest paths from s to v – but we only need one.
2. Any prefix of a shortest path is a shortest paths
   - If the shortest paths from s to v ends in the arc uv, then the sub-path to u is a shortest path from s to u
3. Don't need to consider cycles/loops (shortcut)

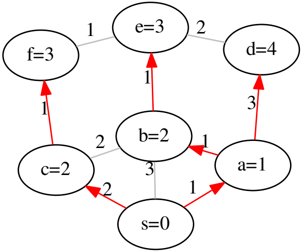
Example



Example graph – We'll find the out-tree and distances from s afterwards, but the result is shown now

Example





Example graph – We'll find the out-tree and distances from s afterwards, but the result is shown now

Shortest path trees

- Representing the shortest paths efficiently (small space, quick information extraction)
- Logically we think of the representation as a tree
- Physically we can use the following:
  - Array distanceTo[n] storing the distance from s to every vertex
  - Array parentOf[n] storing the last vertex u in the shortest path from s to the vertex v
- Reconstructing path – walk backwards along parentOf

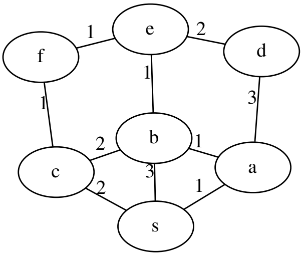
Shortest path trees

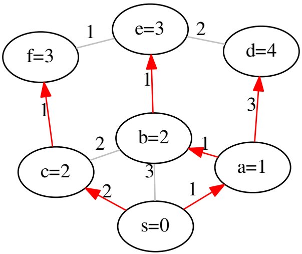
- Representing the shortest paths efficiently (small space, quick information extraction)
- Logically we think of the representation as a tree
- Physically we can use the following:
  - Array distanceTo[n] storing the distance from s to every vertex
  - Array parentOf[n] storing the last vertex u in the shortest path from s to the vertex v
- Reconstructing path – walk backwards along parentOf

Shortest path trees

- Representing the shortest paths efficiently (small space, quick information extraction)
- Logically we think of the representation as a tree
- Physically we can use the following:
  - Array distanceTo[n] storing the distance from s to every vertex
  - Array parentOf[n] storing the last vertex u in the shortest path from s to the vertex v
- Reconstructing path – walk backwards along parentOf

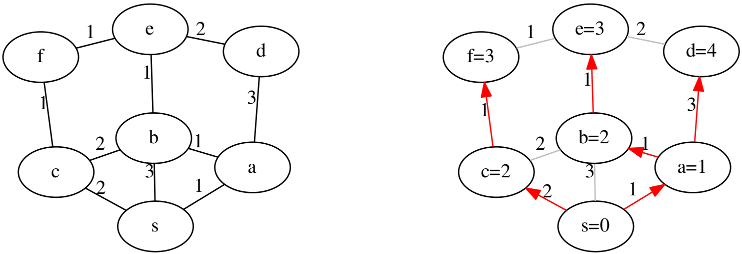
Example





	s	a	b	c	d	e	f
parentOf							
distanceTo							

Example



	s	a	b	c	d	e	f
parentOf	-	s	a	s	a	b	c
distanceTo	0	1	2	2	4	3	3

Correctness criterion for shortest paths

Correctness criterion: Non-improvement

The array distanceTo[v] encodes correct shortest single-source paths if and only if

$$\text{distanceTo[v]} \leq \text{distanceTo[u]} + \text{weight}(uv)$$

holds for every edge uv in the graph.

- Condition is clearly necessary (it must hold)
- Can show sufficiency – see course book ★

Correctness criterion for shortest paths

Correctness criterion: Non-improvement

The array distanceTo[v] encodes correct shortest single-source paths if and only if

$$\text{distanceTo[v]} \leq \text{distanceTo[u]} + \text{weight}(uv)$$

holds for every edge uv in the graph.

- Condition is clearly necessary (it must hold)
- Can show sufficiency – see course book ★

Correctness criterion for shortest paths

Correctness criterion: Non-improvement

The array distanceTo[v] encodes correct shortest single-source paths if and only if

$$\text{distanceTo[v]} \leq \text{distanceTo[u]} + \text{weight}(uv)$$

holds for every edge *uv* in the graph.

- Condition is clearly necessary (it must hold)
- Can show sufficiency – see course book ★

Dijkstra's algorithm

Warm-up: No weights

- Let's assume:
  1. Directed graph
  2. Single-source shortest paths problem
  3. Unit-weight case (no weights)
- Problem easiest solved by:

Warm-up: No weights

- Let's assume:
    1. Directed graph
    2. Single-source shortest paths problem
    3. Unit-weight case (no weights)
- Problem easiest solved by:

Warm-up: No weights

- Let's assume:
    1. Directed graph
    2. Single-source shortest paths problem
    3. Unit-weight case (no weights)
- Problem easiest solved by: BFS

Reminder: Breadth-first search

- Easier algorithm with unit weights only: BFS
- Initialise:
  1. All vertices unmarked
  2. Array distanceTo[n] initialised to ∞
  3. Array parentOf[n] initialised with null
- Start BFS from vertex s:
  1. Create empty Queue q
  2. Mark s, add to queue
  3. distanceTo[s]=0
  4. Until q is empty:

     4.1 Dequeue vertex u from queue

     4.2 For every unmarked neighbour v of u:

     Mark v, add to queue

     Set parentOf[v]=u

     Set distanceTo[v]=distanceTo[u]+1

Reminder: Breadth-first search

- Easier algorithm with unit weights only: BFS
- Initialise:
  1. All vertices unmarked
  2. Array distanceTo[n] initialised to ∞
  3. Array parentOf[n] initialised with null
- Start BFS from vertex s:
  1. Create empty Queue q
  2. Mark s, add to queue
  3. distanceTo[s]=0
  4. Until q is empty:
     - 4.1 Dequeue vertex u from queue
     - 4.2 For every unmarked neighbour v of u:
       - Mark v, add to queue
       - Set parentOf[v]=u
       - Set distanceTo[v]=distanceTo[u]+1

Reminder: Breadth-first search

- Easier algorithm with unit weights only: BFS
- Initialise:
  1. All vertices unmarked
  2. Array distanceTo[n] initialised to ∞
  3. Array parentOf[n] initialised with null
- Start BFS from vertex s:
  1. Create empty Queue q
  2. Mark s, add to queue
  3. distanceTo[s]=0
  4. Until q is empty:
     - 4.1 Dequeue vertex u from queue
     - 4.2 For every unmarked neighbour v of u:
       Mark v, add to queue
       Set parentOf[v]=u
       Set distanceTo[v]=distanceTo[u]+1

Reminder: Breadth-first search

- Easier algorithm with unit weights only: BFS
- Initialise:
  1. All vertices unmarked
  2. Array distanceTo[n] initialised to ∞
  3. Array parentOf[n] initialised with null
- Start BFS from vertex s:
  1. Create empty Queue q
  2. Mark s, add to queue
  3. distanceTo[s]=0
  4. Until q is empty:
     - 4.1 Dequeue vertex u from queue
     - 4.2 For every unmarked neighbour v of u:
       Mark v, add to queue
       Set parentOf[v]=u
       Set distanceTo[v]=distanceTo[u]+1

Dijkstra: Idea and pseudocode

- Idea:
  - Modify BFS to visit closest vertices first
  - Paths with more hops can still be shorter – vertices may overtake each other in the queue
- Implementation: Three vertex types:
  1. Marked vertices (finished – path locked)
  2. Queued vertices (seen, but not finished yet)
  3. Unseen vertices (far away/other components)
- Main loop:
  1. Pull closest vertex u from queue
  2. Consider all edges uv incident with u
  3. Push any newly discovered vertex v into queue
  4. Update any already queued vertex v if uv gives a shorter path

Dijkstra: Idea and pseudocode

- Idea:
  - Modify BFS to visit closest vertices first
  - Paths with more hops can still be shorter – vertices may overtake each other in the queue
- Implementation: Three vertex types:
  1. Marked vertices (finished – path locked)
  2. Queued vertices (seen, but not finished yet)
  3. Unseen vertices (far away/other components)
- Main loop:
  1. Pull closest vertex u from queue
  2. Consider all edges uv incident with u
  3. Push any newly discovered vertex v into queue
  4. Update any already queued vertex v if uv gives a shorter path

Dijkstra: Idea and pseudocode

- Idea:
  - Modify BFS to visit closest vertices first
  - Paths with more hops can still be shorter – vertices may overtake each other in the queue
- Implementation: Three vertex types:
  1. Marked vertices (finished – path locked)
  2. Queued vertices (seen, but not finished yet)
  3. Unseen vertices (far away/other components)
- Main loop:
  1. Pull closest vertex u from queue
  2. Consider all edges uv incident with u
  3. Push any newly discovered vertex v into queue
  4. Update any already queued vertex v if uv gives a shorter path

Update step

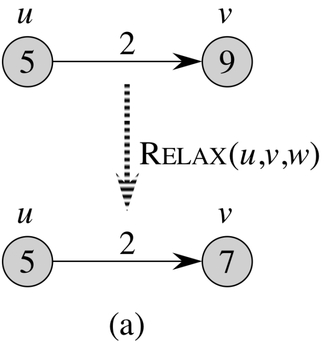


Illustration of update step (called "Relax" above)

Dijkstra: code

Very close to Advanced Prim:

- Data structures:
    1. Min-Priority Queue pq containing queued vertices
    2. parentOf[v] array encoding shortest-path tree
    3. distanceTo[v] array
- Main loop:
    1. Select initial vertex v, call pq.insert(v,0)
    2. Until pq is empty:
        2.1 u=pq.deleteMin(); mark u
        2.2 For every neighbour v of u:
            Call Update(v, uv)

Dijkstra: code

Very close to Advanced Prim:

- Data structures:
  1. Min-Priority Queue pq containing queued vertices
  2. parentOf[v] array encoding shortest-path tree
  3. distanceTo[v] array
- Main loop:
  1. Select initial vertex v, call pq.insert(v,0)
  2. Until pq is empty:

     2.1 u=pq.deleteMin(); mark u

     2.2 For every neighbour v of u:

     Call Update(v, uv)

Dijkstra: Code (final part)

Very close to Advanced Prim:

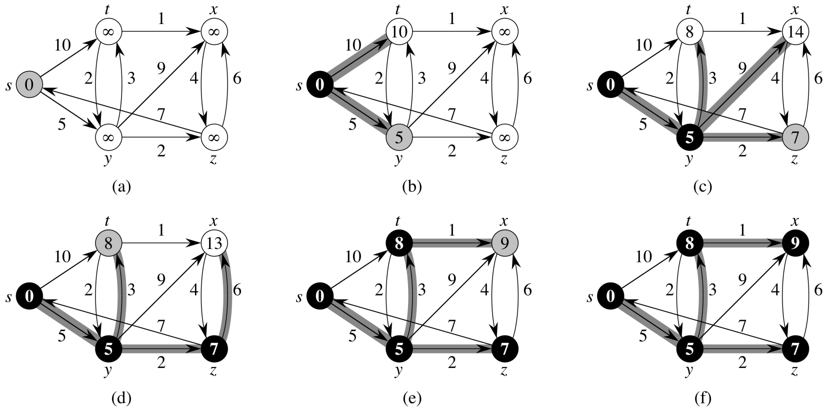
- Data structures:
  1. Min-Priority Queue pq containing queued vertices
  2. parentOf[v] array encoding shortest-path tree
  3. distanceTo[v] array
- Main loop: Pull closest vertex u, make calls Update(v,uv)
- Support code: Update(Vertex v, Edge uv):
  1. Let newDistance = distanceTo[u] + weight(uv)
  2. If *v* marked: return, do nothing
  3. If *v* not in pq:
     - 3.1 parentOf[v] = u
     - 3.2 distanceTo[v] = newDistance
     - 3.3 pq.insert(v, newDistance)
  4. If newDistance < pq.currentValue(v):
     - 4.1 parentOf[v] = u
     - 4.2 distanceTo[v] = newDistance
     - 4.3 pq.decreaseValue(v, newDistance)

Dijkstra: Code (final part)

Very close to Advanced Prim:

- Data structures:
  1. Min-Priority Queue pq containing queued vertices
  2. parentOf[v] array encoding shortest-path tree
  3. distanceTo[v] array
- Main loop: Pull closest vertex u, make calls Update(v,uv)
- Support code: Update(Vertex v, Edge uv):
  1. Let newDistance = distanceTo[u] + weight(uv)
  2. If v marked: return, do nothing
  3. If v not in pq:
     - 3.1 parentOf[v] = u
     - 3.2 distanceTo[v] = newDistance
     - 3.3 pq.insert(v, newDistance)
  4. If newDistance < pq.currentValue(v):
     - 4.1 parentOf[v] = u
     - 4.2 distanceTo[v] = newDistance
     - 4.3 pq.decreaseValue(v, newDistance)

Illustration



Dijkstra in action: black=marked; gray=current; ∞=unseen
Thick edges=Best known path at the time

Dijkstra: Correctness

Suppose we added arc uv in the last iteration.
We know that the array distanceTo[v] encodes correct shortest single-source paths if and only if

$$\text{distanceTo[v]} \leq \text{distanceTo[u]} + \text{weight}(uv)$$

holds for every edge uv in the graph.
Note that the inequality holds. QED.

Dijkstra: Correctness

Suppose we added arc uv in the last iteration.
We know that the array distanceTo[v] encodes correct shortest single-source paths if and only if

$$\text{distanceTo[v]} \leq \text{distanceTo[u]} + \text{weight}(uv)$$

holds for every edge uv in the graph.
Note that the inequality holds. QED.

Dijkstra: Correctness

Suppose we added arc *uv* in the last iteration.
We know that the array distanceTo[v] encodes correct shortest single-source paths if and only if

$$\text{distanceTo[v]} \leq \text{distanceTo[u]} + \text{weight}(uv)$$

holds for every edge *uv* in the graph.
Note that the inequality holds. QED.

Dijkstra: Complexity

Running time of Dijkstra

Dijkstra's algorithm (as described here) has a running time of

- $\mathcal{O}(|E| \log |V|)$ if a "normal" heap+hash implementation of Min-Priority Queue+decreaseValue is used
- $\mathcal{O}(|E| + |V| \log |V|)$ if "theoretical" Fibonacci heaps are used

- The algorithm performs:
  1. $|V|$ insert operations
  2. $|V|$ deleteMin operations
  3. $|E|$ decreaseValue operations (at most)
- There are at most $|V| = n$ entries in pq
- Data structure profiles:
  - Binary heaps: $\mathcal{O}(\log n)$ for insert, deleteMin, decreaseValue
  - Fibonacci: $\mathcal{O}(\log n)$ for insert, deleteMin;
    $\mathcal{O}(1)$ on average over time (amortized) for decreaseValue

Dijkstra: Complexity

Running time of Dijkstra

Dijkstra's algorithm (as described here) has a running time of

- $\mathcal{O}(|E| \log |V|)$ if a "normal" heap+hash implementation of Min-Priority Queue+decreaseValue is used
- $\mathcal{O}(|E| + |V| \log |V|)$ if "theoretical" Fibonacci heaps are used

- The algorithm performs:
  1. $|V|$ insert operations
  2. $|V|$ deleteMin operations
  3. $|E|$ decreaseValue operations (at most)
- There are at most $|V| = n$ entries in pq
- Data structure profiles:
  - Binary heaps: $\mathcal{O}(\log n)$ for insert, deleteMin, decreaseValue
  - Fibonacci: $\mathcal{O}(\log n)$ for insert, deleteMin;
    $\mathcal{O}(1)$ on average over time (amortized) for decreaseValue

Dijkstra: Complexity

Running time of Dijkstra

Dijkstra's algorithm (as described here) has a running time of

- $\mathcal{O}(|E| \log |V|)$ if a "normal" heap+hash implementation of Min-Priority Queue+decreaseValue is used
- $\mathcal{O}(|E| + |V| \log |V|)$ if "theoretical" Fibonacci heaps are used

- The algorithm performs:
  1. $|V|$ insert operations
  2. $|V|$ deleteMin operations
  3. $|E|$ decreaseValue operations (at most)
- There are at most $|V| = n$ entries in pq
- Data structure profiles:
  - Binary heaps: $\mathcal{O}(\log n)$ for insert, deleteMin, decreaseValue
  - Fibonacci: $\mathcal{O}(\log n)$ for insert, deleteMin; $\mathcal{O}(1)$ on average over time (amortized) for decreaseValue

Dijkstra: Complexity

Running time of Dijkstra

Dijkstra's algorithm (as described here) has a running time of

- $\mathcal{O}(|E| \log |V|)$ if a "normal" heap+hash implementation of Min-Priority Queue+decreaseValue is used
- $\mathcal{O}(|E| + |V| \log |V|)$ if "theoretical" Fibonacci heaps are used

- The algorithm performs:
  1. $|V|$ insert operations
  2. $|V|$ deleteMin operations
  3. $|E|$ decreaseValue operations (at most)
- There are at most $|V| = n$ entries in pq
- Data structure profiles:
  - Binary heaps: $\mathcal{O}(\log n)$ for insert, deleteMin, decreaseValue
  - Fibonacci: $\mathcal{O}(\log n)$ for insert, deleteMin;
    $\mathcal{O}(1)$ on average over time (amortized) for decreaseValue

Dijkstra: Complexity

Running time of Dijkstra

Dijkstra's algorithm (as described here) has a running time of

- $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$ if we did not use priority queues (original version)

- Original version (1956)[1]
  - insert is mark vertex as queued, $\mathcal{O}(1)$ time
  - deleteMin is scan through all vertices, $\mathcal{O}(|V|)$ time
  - decreaseValue is reassign distanceTo array, $\mathcal{O}(1)$ time
- $(|V| \cdot \mathcal{O}(1)) + (|V| \cdot \mathcal{O}(|V|)) + (|E| \cdot \mathcal{O}(1)) = \mathcal{O}(|V|^2 + |E|)$
- Dijkstra was a strong motivation for developing fast decreaseValue operations

[1] Heapsort was published 1964, AVL trees 1962

Dijkstra: Complexity

Running time of Dijkstra

Dijkstra's algorithm (as described here) has a running time of

- $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$ if we did not use priority queues (original version)

- Original version (1956)[1]
  - insert is mark vertex as queued, $\mathcal{O}(1)$ time
  - deleteMin is scan through all vertices, $\mathcal{O}(|V|)$ time
  - decreaseValue is reassign distanceTo array, $\mathcal{O}(1)$ time
- $(|V| \cdot \mathcal{O}(1)) + (|V| \cdot \mathcal{O}(|V|)) + (|E| \cdot \mathcal{O}(1)) = \mathcal{O}(|V|^2 + |E|)$
- Dijkstra was a strong motivation for developing fast decreaseValue operations

[1] Heapsort was published 1964, AVL trees 1962

Dijkstra: Complexity

Running time of Dijkstra

Dijkstra's algorithm (as described here) has a running time of

- $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$ if we did not use priority queues (original version)

- Original version (1956)[1]
  - insert is mark vertex as queued, $\mathcal{O}(1)$ time
  - deleteMin is scan through all vertices, $\mathcal{O}(|V|)$ time
  - decreaseValue is reassign distanceTo array, $\mathcal{O}(1)$ time
- $(|V| \cdot \mathcal{O}(1)) + (|V| \cdot \mathcal{O}(|V|)) + (|E| \cdot \mathcal{O}(1)) = \mathcal{O}(|V|^2 + |E|)$
- Dijkstra was a strong motivation for developing fast decreaseValue operations

[1] Heapsort was published 1964, AVL trees 1962

Dijkstra: Complexity

Running time of Dijkstra

Dijkstra's algorithm (as described here) has a running time of

- $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$ if we did not use priority queues (original version)

- Original version (1956)[1]
  - insert is mark vertex as queued, $\mathcal{O}(1)$ time
  - deleteMin is scan through all vertices, $\mathcal{O}(|V|)$ time
  - decreaseValue is reassign distanceTo array, $\mathcal{O}(1)$ time
- $(|V| \cdot \mathcal{O}(1)) + (|V| \cdot \mathcal{O}(|V|)) + (|E| \cdot \mathcal{O}(1)) = \mathcal{O}(|V|^2 + |E|)$
- Dijkstra was a strong motivation for developing fast decreaseValue operations

[1] Heapsort was published 1964, AVL trees 1962

An example of Dijkstra's use

Go to Slide 9.

★Other applications

- Paths in directed acyclic graphs
  - Can compute in $\mathcal{O}(|E| + |V|)$ time
  - Can also find longest paths (normally difficult)
- Application: job scheduling
  - Collection of jobs with processing times and precedence
  - Nodes: Job i starts and Job i ends
  - Arcs Start(i) $\to$ End(i), weight = processing time of job
  - Arcs End(i) $\to$ Start(j), weight 0: Precedence
  - Longest path determines critical path – minimum processing time
- Can even use negative weight arc Start(i) $\to$ End(j) to say "Job j must end at the earliest X minutes before job i starts"
  - I.e., "no cooldown" – job i starts not-too-long after job j ends

★Other applications

- Paths in directed acyclic graphs
  - Can compute in $\mathcal{O}(|E| + |V|)$ time
  - Can also find longest paths (normally difficult)
- Application: job scheduling
  - Collection of jobs with processing times and precedence
  - Nodes: Job i starts and Job i ends
  - Arcs Start(i) $\to$ End(i), weight $=$ processing time of job
  - Arcs End(i) $\to$ Start(j), weight 0: Precedence
  - Longest path determines critical path – minimum processing time
- Can even use negative weight arc Start(i) $\to$ End(j) to say "Job j must end at the earliest X minutes before job i starts"
  - I.e., "no cooldown" – job i starts not-too-long after job j ends

★Other applications

- Paths in directed acyclic graphs
  - Can compute in $\mathcal{O}(|E| + |V|)$ time
  - Can also find longest paths (normally difficult)
- Application: job scheduling
  - Collection of jobs with processing times and precedence
  - Nodes: Job i starts and Job i ends
  - Arcs Start(i) $\rightarrow$ End(i), weight = processing time of job
  - Arcs End(i) $\rightarrow$ Start(j), weight 0: Precedence
  - Longest path determines critical path – minimum processing time
- Can even use negative weight arc Start(i) $\rightarrow$ End(j) to say "Job j must end at the earliest X minutes before job i starts"
  - I.e., "no cooldown" – job i starts not-too-long after job j ends

★Other applications

- Paths in directed acyclic graphs
  - Can compute in $\mathcal{O}(|E| + |V|)$ time
  - Can also find longest paths (normally difficult)
- Application: job scheduling
  - Collection of jobs with processing times and precedence
  - Nodes: Job i starts and Job i ends
  - Arcs $\text{Start}(i) \to \text{End}(i)$, weight = processing time of job
  - Arcs $\text{End}(i) \to \text{Start}(j)$, weight 0: Precedence
  - Longest path determines critical path – minimum processing time
- Can even use negative weight arc $\text{Start}(i) \to \text{End}(j)$ to say "Job j must end at the earliest X minutes before job i starts"
  - I.e., "no cooldown" – job i starts not-too-long after job j ends