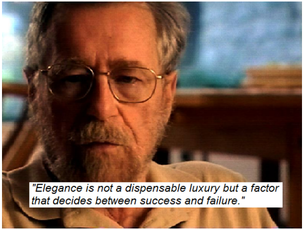

THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Algorithms and Data Structures

Lecture 16 Distances in Weighted Graphs

Jiamou Liu
The University of Auckland



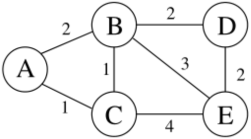
Weighted Graphs

In real appliations, we need to find distances in weighted graphs, that are graph whose edges have integer weights.

In real appliations, we need to find distances in weighted graphs, that are graph whose edges have integer weights.

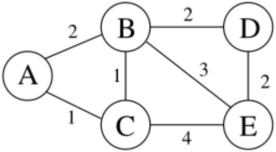
Weighted Graph

A weighted graph is $G = (V, E, w)$ where (V, E) is an undirected graph and $w : E \to \mathbb{Z}$ is a weight function that assigns each edge with an integer weight.



Weighted Graph Representation

We extend adjacency matrix or adjacency list representations to weighted graphs.

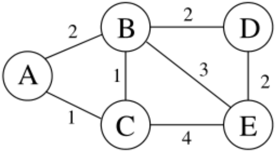


$$
\begin{bmatrix}
0 & 2 & 1 & \infty & \infty \\
2 & 0 & 1 & 2 & 3 \\
1 & 1 & 0 & \infty & 4 \\
\infty & 2 & \infty & 0 & 2 \\
\infty & 3 & 4 & 2 & 0
\end{bmatrix}
$$

Adjacency Matrix representation of weighted graphs

Weighted Graph Representation

We extend adjacency matrix or adjacency list representations to weighted graphs.



5
B:2, C:1
A:2, C:1, D:2, E:3
A:1, B:1, E:4
B:2, E:2
B:3, D:2, C:4

Adjacency List representation of weighted graphs

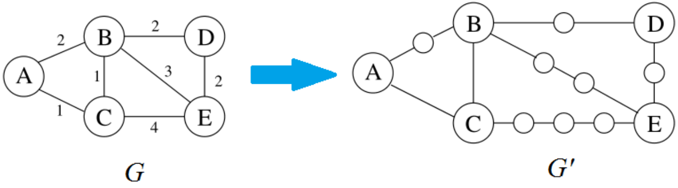
Distances Weighted Graphs

Goal. Compute distances in a weighted graph.

Distances Weighted Graphs

Goal. Compute distances in a weighted graph.

If weights are integers, subdivide edges into a sequence of unit-length edges.



Convert a weighted graph G into an unweighted graph G'

However this is very inefficient, as the time complexity of BFS would depend on the sum of all weights.

Goal. Compute distances in a weighted graph more efficiently.

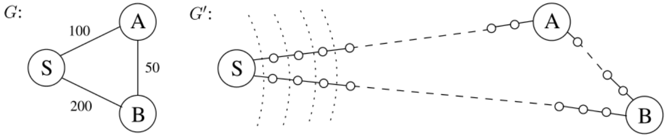
Shortest Path Problem (Single-Sourced)

- **INPUT**: a weighted graph G and a source node s
- **OUTPUT**: the shortest path from s to all other nodes.

A "Lazy" Way for Finding Distances

Recap from Last Lecture

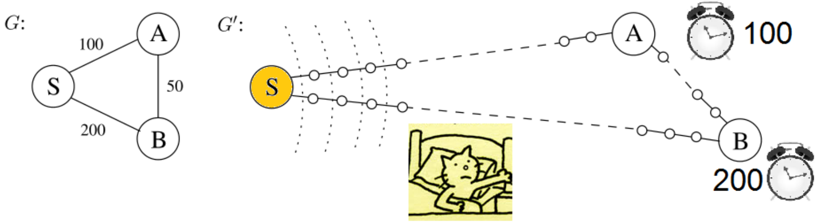
- We can transform a weighted graph into an unweighted one.
- The approach is very inefficient because there could be a large number of auxiliary nodes
- We don't care about the distances on these auxiliary nodes
- We could just "go to sleep" when BFS visits these auxiliary nodes
- But we need to "wake up" when BFS visits an original node



Setting Alarm Clocks

Strategy

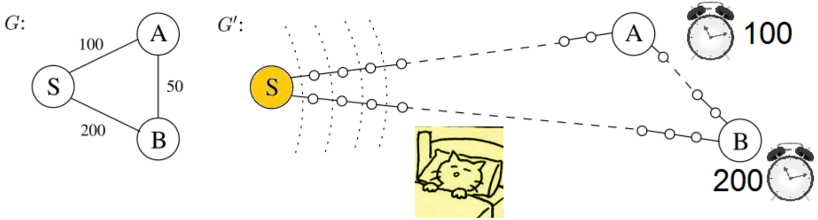
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

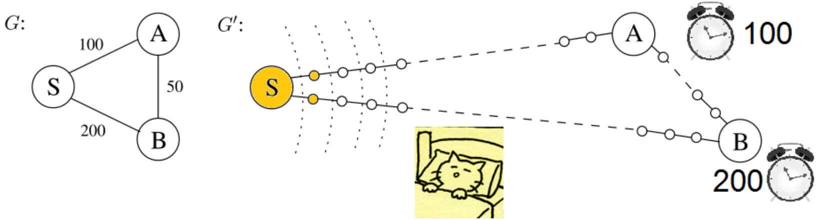
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

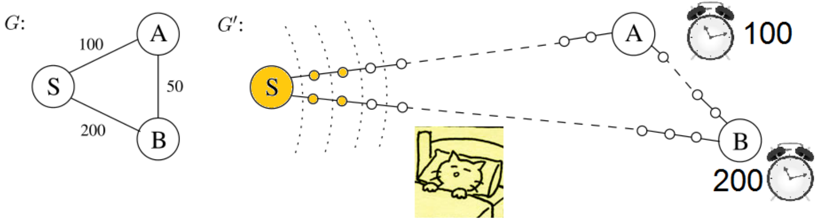
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

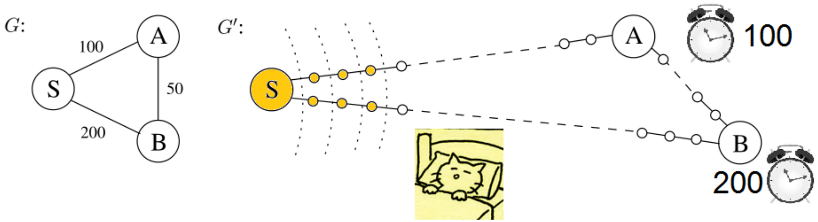
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

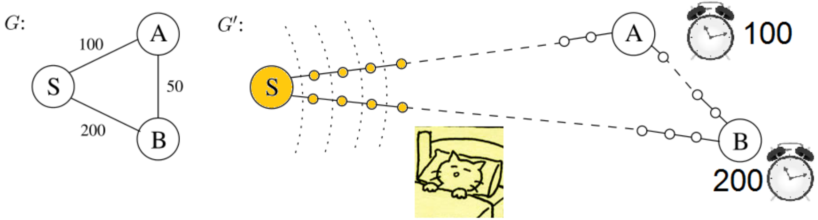
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

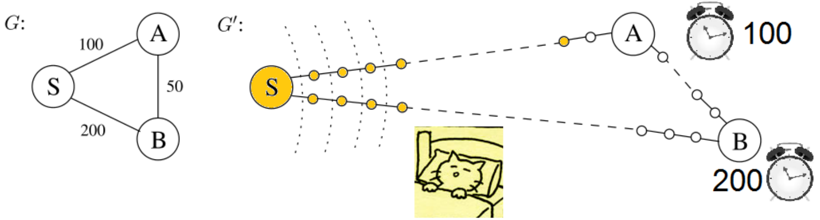
- Maintain an “alarm clock” for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

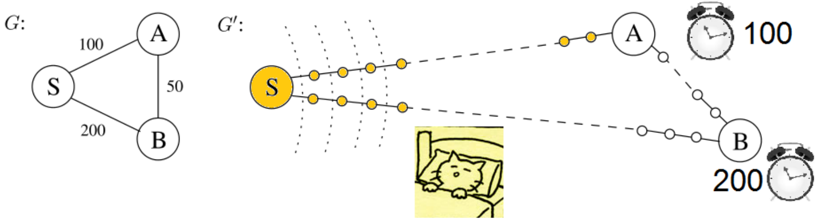
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

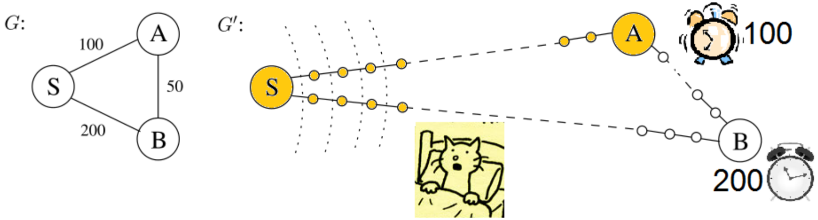
- Maintain an “alarm clock” for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

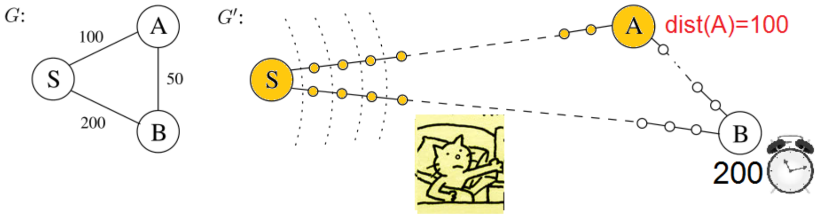
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

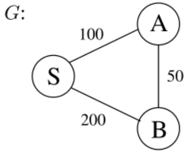
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks

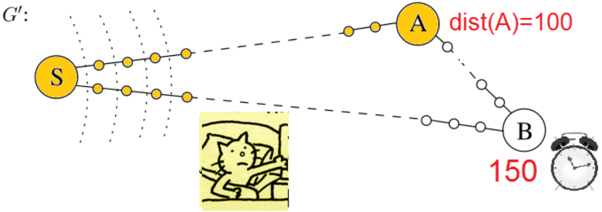


Setting Alarm Clocks

Strategy

- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks

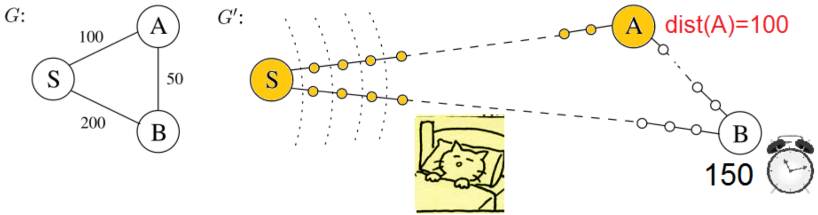




Setting Alarm Clocks

Strategy

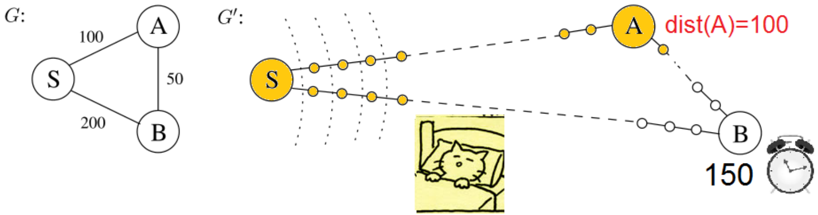
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

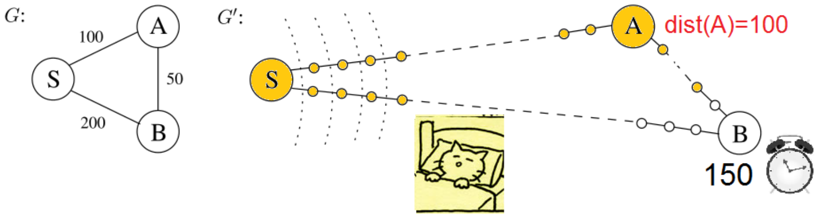
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

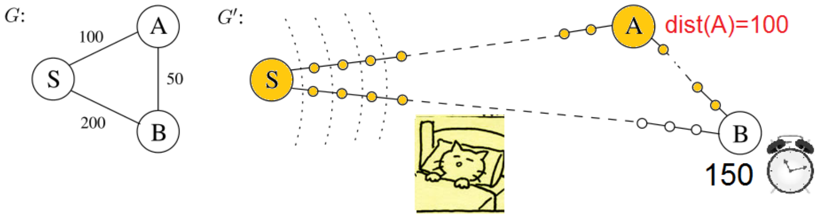
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

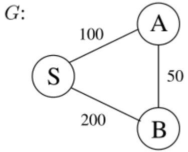
- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks

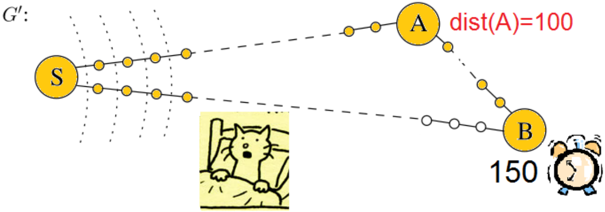


Setting Alarm Clocks

Strategy

- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks

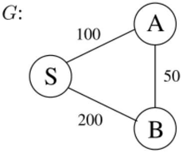


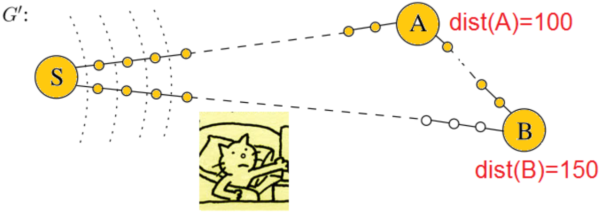


Setting Alarm Clocks

Strategy

- Maintain an "alarm clock" for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks





“Alarm Clock Algorithm”

Simulate the execution of BFS on G' starting from node s.

“Alarm Clock Algorithm”

1. Set an alarm clock for each node for time $w(s, v)$
2. Start BFS on G' and go to sleep
3. **Repeat** the following **until** no more alarm is left:
4.   Whenever an alarm clock goes off, wake up
5.   Pause BFS
6.   Check the current time, say T
7.   **If** this is u's alarm, **then** write $dist(u) = T$
8.   Discard this alarm clock
9.   **For** each neighbor v of u **do**:
10.   **If** v's alarm is set for a time $> T + w(u, v)$,
            then reset it to $T + w(u, v)$
12.   Resume BFS and go back to sleep

Implementing the Alarm Clocks

Question: How do we implement the system of alarm clocks in a computer?

Implementing the Alarm Clocks

Question: How do we implement the system of alarm clocks in a computer?

We need a data structure that is able to:

- Contain a collection of integer keys (i.e. alarm clock times)
- Insert(e, k): Add a new element e with key k to the collection
- DeleteMin(): Return the element with the smallest key, and remove it from the collection
- ResetKey(e, k): Reset the key value of element e to a smaller value k

Implementing the Alarm Clocks

Question: How do we implement the system of alarm clocks in a computer?

We need a data structure that is able to:

- Contain a collection of integer keys (i.e. alarm clock times)
- Insert(e, k): Add a new element e with key k to the collection
- DeleteMin(): Return the element with the smallest key, and remove it from the collection
- ResetKey(e, k): Reset the key value of element e to a smaller value k

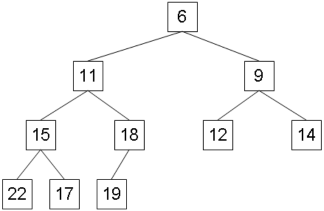
This is a Priority Queue!

Priority Queue

Priority Queues

A priority queue is a data structure that stores a collection of $(element, key)$ pairs where the *key* of an element is an integer value and allows the following operations:

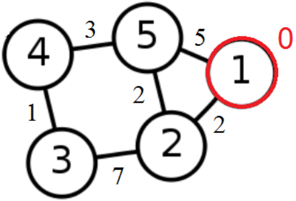
- $Insert(e, k)$: Add a new element e with key k to the collection
- $DeleteMin()$: Return the element with the smallest key, and remove it from the collection
- $ResetKey(e, k)$: Reset the key value of element e to a smaller value k

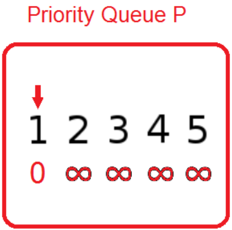


Dijkstra’s Algorithm

Implementing the “Alarm Clocks” using Priority Queue:

- Maintain a priority key set for each node
- Maintain a set of confirmed nodes

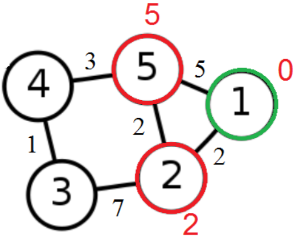


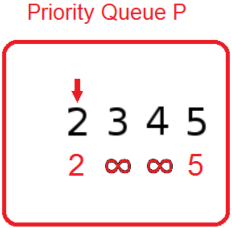


Dijkstra's Algorithm

Implementing the "Alarm Clocks" using Priority Queue:

- Maintain a priority key set for each node
- Maintain a set of confirmed nodes

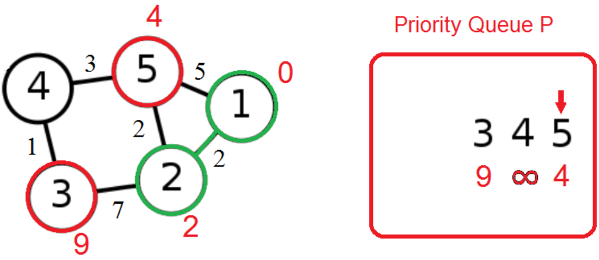




Dijkstra's Algorithm

Implementing the "Alarm Clocks" using Priority Queue:

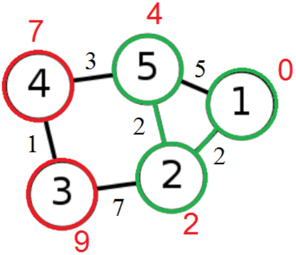
- Maintain a priority key set for each node
- Maintain a set of confirmed nodes

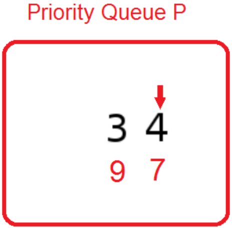


Dijkstra's Algorithm

Implementing the "Alarm Clocks" using Priority Queue:

- Maintain a priority key set for each node
- Maintain a set of confirmed nodes

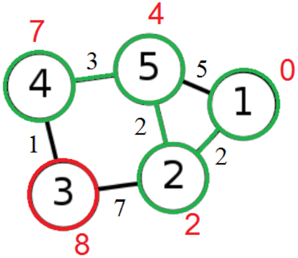


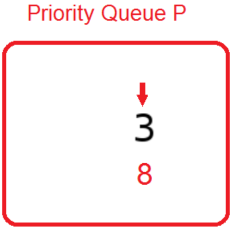


Dijkstra's Algorithm

Implementing the "Alarm Clocks" using Priority Queue:

- Maintain a priority key set for each node
- Maintain a set of confirmed nodes

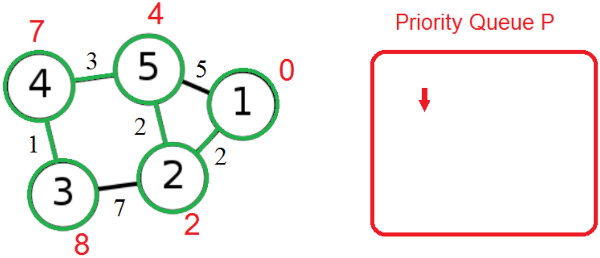




Dijkstra's Algorithm

Implementing the "Alarm Clocks" using Priority Queue:

- Maintain a priority key set for each node
- Maintain a set of confirmed nodes



Algorithm Dijkstra(G, s)

INPUT: A weighted graph G, and a node s
OUTPUT: $dist(v)$ of all nodes v

```
dist(s) ← 0
Initialize a set Reach ← {s}
Initialize a priority queue P containing (s, 0)
for u ∈ V, u ≠ s do
    dist(u) ← ∞
    prev(u) ← null
    P.Insert(u, ∞)
while P is not empty do
    u ← P.DeleteMin()
    Add u to Reach
    for (u, v) ∈ E where v ∉ Reach do
        if dist(u) + w(u, v) < dist(v) do
            dist(v) ← dist(u) + w(u, v)
            P.ResetKey(v, dist(v))
            prev(v) ← u
```

Dijkstra's Algorithm: Correctness

Theorem

Let G be a weighted graph with positive weights only. After running Dijkstra's algorithm on G and a node s in G, $dist(u)$ is the distance from s to u for every node u.

Dijkstra's Algorithm: Correctness

Theorem

Let G be a weighted graph with positive weights only. After running Dijkstra's algorithm on G and a node s in G, $dist(u)$ is the distance from s to u for every node u.

Proof.

We prove the following loop invariant by induction on n:
At the end of the nth iteration of the **while** -loop, we have

(a) there is d such that all nodes in *Reach* are at distance $\leq d$ from s and all nodes outside *Reach* are at distances $\geq d$ from s

(b) for every node u, the value $dist(u)$ is the length of the shortest path from s to u whose intermediate nodes are all in *Reach*.

Note that the above statements imply the theorem.

Proof. (Continued)

Base case: When $n = 0$, prior to running the **while** -loop, the only node in R is s and $d = 0$. Both (a),(b) clearly hold.

Proof. (Continued)

Base case: When $n = 0$, prior to running the **while** -loop, the only node in R is s and $d = 0$. Both (a),(b) clearly hold.

Inductive Step: Suppose after the nth iteration of the **while**-loop, (a),(b) both hold. We would like to show (a),(b) hold after the $(n + 1)$th iteration.

Proof. (Continued)

Base case: When $n = 0$, prior to running the **while** -loop, the only node in R is s and $d = 0$. Both (a),(b) clearly hold.

Inductive Step: Suppose after the nth iteration of the **while**-loop, (a),(b) both hold. We would like to show (a),(b) hold after the $(n+1)$th iteration.

(a) Suppose we have just finished the ith iteration.
By (a), all nodes in R have distance $\leq d$ for some d.
Let $u \notin Reach$ be the node whose distance from s is the next smallest.

- Then the shortest path to u must be $s \rightsquigarrow v \to u$ where $v \in Reach$.
- Thus u must be the node with the minimal key in the priority queue P.
- Hence after the $(n+1)$th iteration, all nodes in $Reach$ have distance $\leq dist(u)$ and all nodes not in $Reach$ have distance $\geq dist(u)$.
- Therefore (a) holds after $(n+1)$th iteration.

Proof. (Continued)

(b) Suppose we have just finished the $(n+1)$th iteration.
Take any $v \notin \mathit{Reach}$ and the shortest path from s to v that passes through only nodes in Reach.

Proof. (Continued)

(b) Suppose we have just finished the $(n+1)$th iteration.
Take any $v \notin Reach$ and the shortest path from s to v that passes through only nodes in $Reach$.

Case 1: The second to last node in this path is not u.
Then $dist(v)$ did not change in the $(n+1)$th iteration.

Case 2: The second to last node in this path is u.
Then $dist(v)$ is changed to $dist(u) + w(u, v)$.

Proof. (Continued)

(b) Suppose we have just finished the $(n+1)$th iteration.
Take any $v \notin \mathit{Reach}$ and the shortest path from s to v that passes through only nodes in Reach.

Case 1: The second to last node in this path is not u.
Then $\mathit{dist}(v)$ did not change in the $(n+1)$th iteration.

Case 2: The second to last node in this path is u.
Then $\mathit{dist}(v)$ is changed to $\mathit{dist}(u) + w(u, v)$.

- In both cases (b) is satisfied.
- Thus (b) holds after the $(n+1)$th iteration.
- Therefore the theorem is proved. □

Dijkstra's Algorithm: Complexity

Note: The running time of Dijkstra's algorithm depends on the running time of priority queue implementations.

- Each node is inserted to the priority queue once
- For each edge, we may reset the key of an element in the priority queue
- Each node is deleted from the priority queue once

Therefore

- Let $T_{in}(n)$ be the time it takes to insert elements to the priority queue
- Let $T_{re}(n)$ be the time it takes to reset key for an element in the priority queue
- Let $T_{de}(n)$ be the time it takes to delete min element from the priority queue

Dijkstra's Algorithm: Complexity

Note: The running time of Dijkstra's algorithm depends on the running time of priority queue implementations.

- Each node is inserted to the priority queue once
- For each edge, we may reset the key of an element in the priority queue
- Each node is deleted from the priority queue once

Therefore

- Let $T_{in}(n)$ be the time it takes to insert elements to the priority queue
- Let $T_{re}(n)$ be the time it takes to reset key for an element in the priority queue
- Let $T_{de}(n)$ be the time it takes to delete min element from the priority queue

Running time $nT_{in}(n) + mT_{re}(n) + nT_{de}(n)$

Dijkstra's Algorithm: Complexity

Note: The running time of Dijkstra's algorithm depends on the running time of priority queue implementations.

- Each node is inserted to the priority queue once
- For each edge, we may reset the key of an element in the priority queue
- Each node is deleted from the priority queue once

Therefore

- Let $T_{in}(n)$ be the time it takes to insert elements to the priority queue
- Let $T_{re}(n)$ be the time it takes to reset key for an element in the priority queue
- Let $T_{de}(n)$ be the time it takes to delete min element from the priority queue

Running time $nT_{in}(n) + mT_{re}(n) + nT_{de}(n)$
We now look at some standard priority queues.

Different Priority Queues

Which one is better for Dijkstra's Algorithm?

- Linked List:
- Binary Heap:

Different Priority Queues

Which one is better for Dijkstra's Algorithm?

- Linked List: $O(n^2)$

- Binary Heap: $O((m + n) \log n)$

Different Priority Queues

Which one is better for Dijkstra's Algorithm?

- Linked List: $O(n^2)$

  Preferred when there are a lot of edges, i.e., $m \geq \frac{n^2}{\log n}$

- Binary Heap: $O((m+n)\log n)$

Different Priority Queues

Which one is better for Dijkstra's Algorithm?

- Linked List: $O(n^2)$

  Preferred when there are a lot of edges, i.e., $m \geq \frac{n^2}{\log n}$

- Binary Heap: $O((m+n)\log n)$

  Preferred when there are not a lot of edges, i.e., $m < \frac{n^2}{\log n}$

Different Priority Queues

Which one is better for Dijkstra's Algorithm?

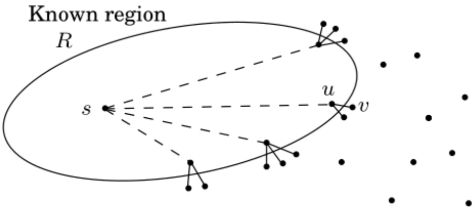
- Linked List: $O(n^2)$

  Preferred when there are a lot of edges, i.e., $m \geq \frac{n^2}{\log n}$
- Binary Heap: $O((m+n)\log n)$

  Preferred when there are not a lot of edges, i.e., $m < \frac{n^2}{\log n}$
- Fibonacci Heap: $O(n \log n + m)$

  Better asymptotic complexity, but complicated to implement. (Not covered in this course.)

Graph Exploration



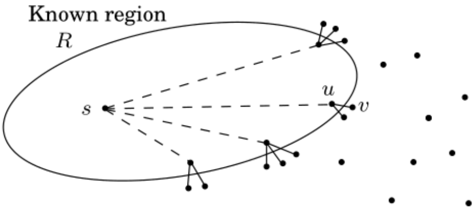
Exploring Graphs

A graph exploration algorithm traverses the graph:

- The algorithm maintains a known region of nodes
- Each time it picks an edge that goes out from the known region, exploring a node outside and expanding its known region
- It stops when no more edge can be explored

The order in which new edges are picked determines the type of algorithm.

Graph Exploration



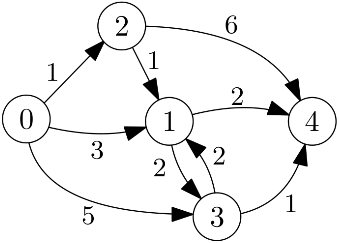
Exploring Graphs

- DFS picks edges based on a stack order
  $\Rightarrow$ Exploration is as deep as possible
- BFS picks edges based on a queue order
  $\Rightarrow$ Exploration is as broad as possible (hence revealing distances)
- Dijkstra's algorithm picks edges based on a priority order (on a weighted graph
  $\Rightarrow$ Exploration is as broad as possible (hence revealing distances)

Summary

- Computing distance in unweighted graphs:
  - Breadth-first search (BFS) algorithm
  - Queue implementation of the BFS algorithm
- Computing distance in weighted graphs:
  - Dijkstra's algorithm
  - Priority-queue implementation of Dijkstra's algorithm
- A unifying framework of graph exploration

Question 1. On the graph below, run Dijkstra's algorithm starting from node 0. Draw the content of the priority queue at each step, and the shortest paths found from 0 to all nodes.



Exercise

Question 2. Suppose you would like to find the longest path from a given node to other nodes in a weighted directed acyclic graph[1]. Someone suggests that maybe we can try modifying Dijkstra's algorithm, so that instead of using a min-heap (as we do for dijkstra's algorithm), we use a *max-heap*, i.e., a priority queue where we have `DeleteMax` operation that returns and removes the maximum element, and update the value $dist(v)$ for a node v whenever we find a *longer* path to v. Do you think this algorithm would correctly compute the longest distance from the starting node to other nodes in a dag? Explain why.

[1] Why must you restrict to only acyclic graphs?


THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND