Algorithm Design and Analysis

Day 6
Distances in Graphs

2015, AUT-CJLU

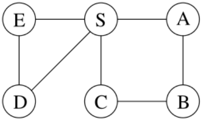
Part I: Breadth First Search

Traversing a Graph

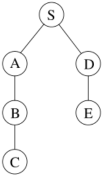
Uses for Graph Traversals

- Searching (DFS)
- Reachability (DFS)
- Decomposing graphs (DFS)
- Calculating distances (DFS is not suitable)

Graph G



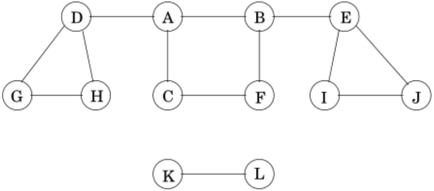
The DFS Tree of G



Distances Between Nodes

Recall: Let G be a graph

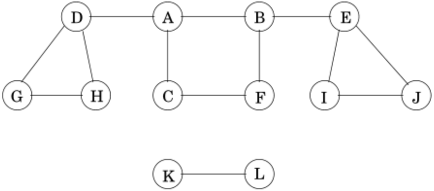
- The length of a path is the number of edges ("steps") in it.
- The distance from a node u to v, $dist(u, v)$, is the length of the shortest path from u to v. If no path exist, then $dist(u, v) = \infty$.



Distances Between Nodes

Recall: Let G be a graph

- The length of a path is the number of edges (“steps”) in it.
- The distance from a node u to v, $dist(u, v)$, is the length of the shortest path from u to v. If no path exist, then $dist(u, v) = \infty$.



- Distance between D and J: 4
- Distance between A and K: ∞

Distances Between Nodes

Distance Problem

INPUT: a graph G, and two nodes u, v
OUTPUT: the distance from u to v.

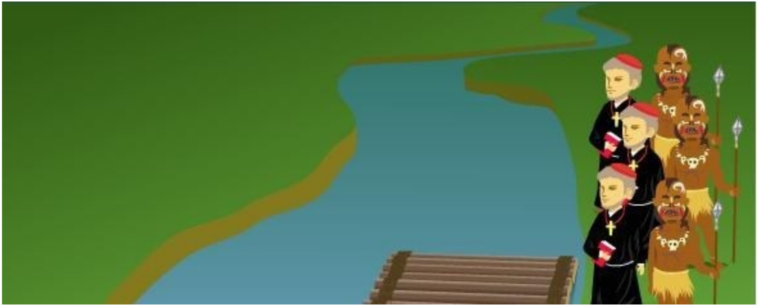
Shortest Path Problem

INPUT: a graph G, and two nodes u, v
OUTPUT: the shortest path from u to v.

Case Study: Missionary and Cannibals

Missionaries and Cannibals

There are 3 missionaries and 3 cannibals coming to a river with only one boat that can hold only 2 people. At any instance the number of cannibals cannot be more than the number of missionaries. How can all the people cross the river using the least number of boat rides?



Case Study: Missionary and Cannibals

Convert the problem into a graph problem:

- We call a time stamp of the current situation a configuration.
- A configuration states how many missionaries and cannibals on each bank, and the location of the boat.

  E.g. $(MC\bullet, MMCC)$ indicates 1 missionary + 1 cannibal on left bank, 2 missionaries + 2 cannibals on right bank, boat on left bank

Case Study: Missionary and Cannibals

Convert the problem into a graph problem:

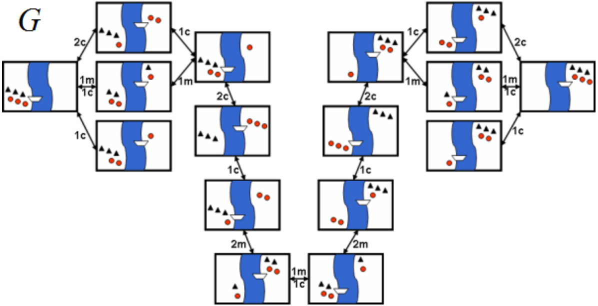
- We call a time stamp of the current situation a configuration.
- A configuration states how many missionaries and cannibals on each bank, and the location of the boat.

  E.g. $(MC\bullet, MMCC)$ indicates 1 missionary + 1 cannibal on left bank, 2 missionaries + 2 cannibals on right bank, boat on left bank
- Define a graph $G = (V, E)$ where

  - The nodes are all configurations

  - Two nodes are connected by an edge if from one node (configuration) the six people can move to the other node (configuration) using one boat ride.

Case Study: Missionary and Cannibals



Our goal: Find a shortest path from $(, \bullet MMMCCC)$ to $(MMMCCC\bullet,)$.

Breadth First Search

How to solve the distance and the shortest path problem?

Breadth First Search

How to solve the distance and the shortest path problem?

Breadth First Search Strategy

Traverse nodes by "layers":

1. Visit the start node s
2. Visit all nodes that have distance 1 from s (call them V_1)
3. Visit all nodes that have distance 1 from V_1 (call them V_2)
4. Visit all nodes that have distance 1 from V_2 (call them V_3)
5. … …

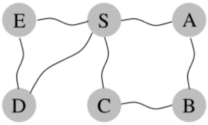
Breadth First Search

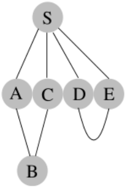
How to solve the distance and the shortest path problem?

Breadth First Search Strategy

Traverse nodes by "layers":

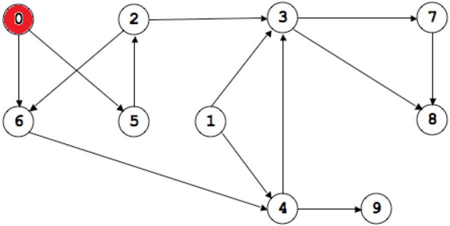
1. Visit the start node s
2. Visit all nodes that have distance 1 from s (call them V_1)
3. Visit all nodes that have distance 1 from V_1 (call them V_2)
4. Visit all nodes that have distance 1 from V_2 (call them V_3)
5. ... ...





Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:
- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.



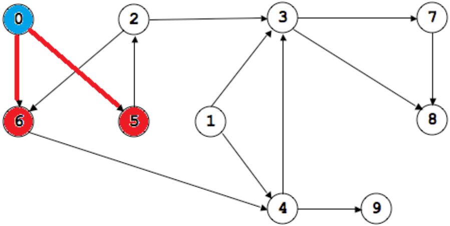
$Q = [0]$

Breadth First Search

Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:

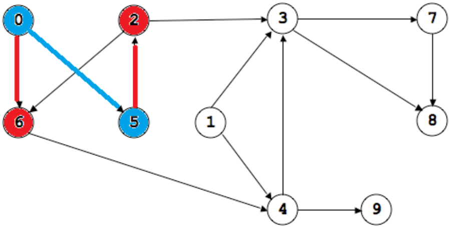
- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.



$Q = [5, 6]$

Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:
- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.

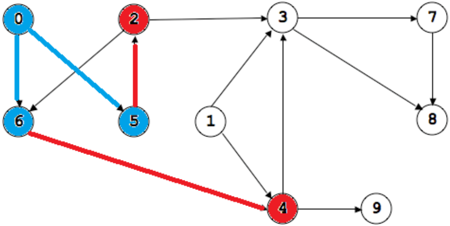


$Q = [6, 2]$

Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:

- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.



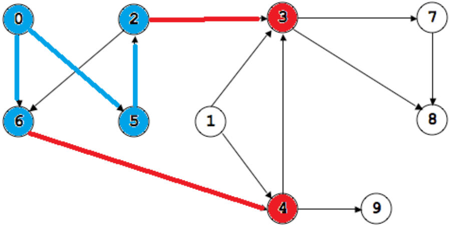
$Q = [2, 4]$

Breadth First Search

Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:

- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.



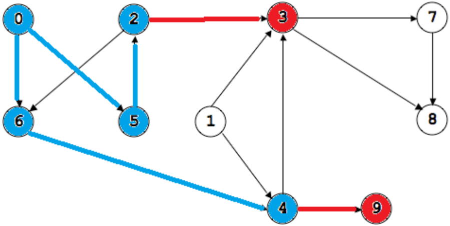
$Q = [4, 3]$

Breadth First Search

Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:

- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.



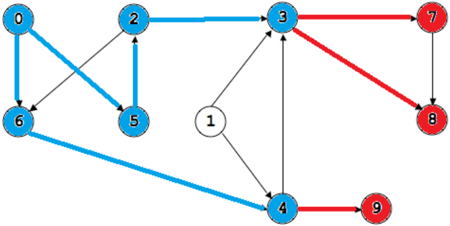
$Q = [3, 9]$

Breadth First Search

Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:

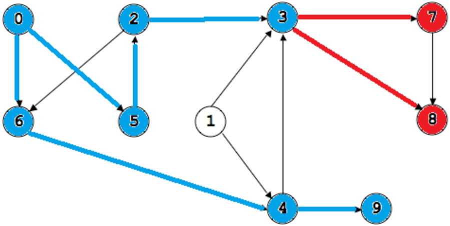
- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.



$Q = [9, 7, 8]$

Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:
- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.



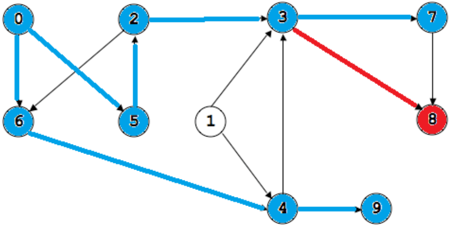
$Q = [7, 8]$

Breadth First Search

Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:

- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.



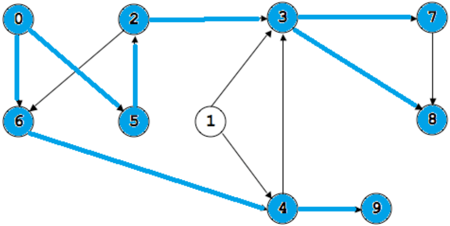
$Q = [8]$

Breadth First Search

Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:

- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.



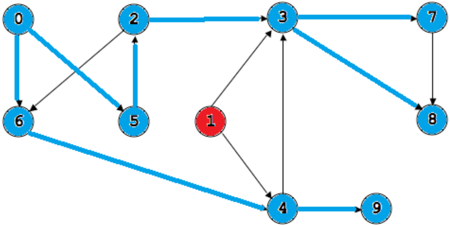
$Q = []$

Breadth First Search

Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:

- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.

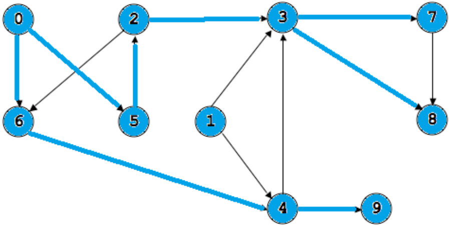


$Q = [1]$

Breadth First Search

Queue implementation of BFS

Maintain a queue of to-be-explored nodes.
At each iteration:
- First finish the first element in the queue; dequeue.
- Then enqueue the out-neighbours of the dequeued element.



$Q = []$

Breadth First Search

Maintain a field $dist(u)$ for every node u.

Procedure bfs_explore(G, s)

INPUT: A graph G and a starting node s
$dist(s) \leftarrow 0$
Create an empty queue Q and enqueue(Q, s)
`while` Q is not empty `do`
    $u \leftarrow$ dequeue(Q) (Note u is first in Q)
    `for` any outgoing edge (u, v) `do`
        `if` $dist(v) = \infty$ `then`
            enqueue(Q, v)
            $dist(v) \leftarrow dist(u) + 1$

Procedure bfs(G)

INPUT: A graph G and a starting node s
OUTPUT: Labeling $dist(u)$ for every u
`for` $u \in V$ `do` $dist(u) \leftarrow \infty$
`for` $u \in V$ `do`
    `if` $dist(u) = \infty$ `then` run bfs_explore(G, u)

Breadth First Search

Theorem.

After running bfs_explore(G, s), the value of $dist(u)$ is the distance from s to u for every node u.

Breadth First Search

Theorem.

After running bfs_explore(G, s), the value of $dist(u)$ is the distance from s to u for every node u.

Proof. We use the notion of known region: Say a node u is in the known region if $dist(u) \neq \infty$.

Breadth First Search

Theorem.

After running bfs_explore(G, s), the value of $dist(u)$ is the distance from s to u for every node u.

Proof. We use the notion of known region: Say a node u is in the known region if $dist(u) \neq \infty$.
We need to prove that all reachable nodes u from s eventually enters the known region, and that $dist(u)$ will be correctly calculated.

Breadth First Search

Theorem.

After running bfs_explore(G, s), the value of $dist(u)$ is the distance from s to u for every node u.

Proof. We use the notion of known region: Say a node u is in the known region if $dist(u) \neq \infty$.
We need to prove that all reachable nodes u from s eventually enters the known region, and that $dist(u)$ will be correctly calculated.

Suppose, for a contradiction, that u is a closest node that will not enter the known region.

Breadth First Search

Theorem.

After running bfs_explore(G, s), the value of $dist(u)$ is the distance from s to u for every node u.

Proof. We use the notion of known region: Say a node u is in the known region if $dist(u) \neq \infty$.
We need to prove that all reachable nodes u from s eventually enters the known region, and that $dist(u)$ will be correctly calculated.

Suppose, for a contradiction, that u is a closest node that will not enter the known region.
Say $dist(u) = k > 0$. Then there is a node v with $dist(v) = k - 1$ and $(v, u) \in E$.

Breadth First Search

Theorem.

After running bfs_explore(G, s), the value of $dist(u)$ is the distance from s to u for every node u.

Proof. We use the notion of known region: Say a node u is in the known region if $dist(u) \neq \infty$.
We need to prove that all reachable nodes u from s eventually enters the known region, and that $dist(u)$ will be correctly calculated.

Suppose, for a contradiction, that u is a closest node that will not enter the known region.
Say $dist(u) = k > 0$. Then there is a node v with $dist(v) = k - 1$ and $(v, u) \in E$.
Then v would enter the known region and all outgoing edges of v will be examined.

Breadth First Search

Theorem.

After running bfs_explore(G, s), the value of $dist(u)$ is the distance from s to u for every node u.

Proof. We use the notion of known region: Say a node u is in the known region if $dist(u) \neq \infty$.
We need to prove that all reachable nodes u from s eventually enters the known region, and that $dist(u)$ will be correctly calculated.

Suppose, for a contradiction, that u is a closest node that will not enter the known region.
Say $dist(u) = k > 0$. Then there is a node v with $dist(v) = k - 1$ and $(v, u) \in E$.
Then v would enter the known region and all outgoing edges of v will be examined.
Thus u will enter the known region and $dist(u)$ will be k.
Contradiction $\square$

Breadth First Search - Complexity

Complexity Analysis

In running a BFS on G:

- Every node u in G may be enqueued and dequeued at most once
- Every edge may be checked at most once

Breadth First Search - Complexity

Complexity Analysis

In running a BFS on G:

- Every node u in G may be enqueued and dequeued at most once
- Every edge may be checked at most once

$\Rightarrow$ The running time of the BFS algorithm is $O(m + n)$.

DFS versus BFS

DFS: The algorithm makes deep but narrow exploration into the graph.

Only retreat when it runs out of new nodes.

Implementation: Use a stack as an auxiliary data structure. Can also be implemented recursively.

Running Time: $O(m + n)$.

Applications: This could be used for analyzing reachability, linearizability, connectedness.

DFS versus BFS

DFS: The algorithm makes deep but narrow exploration into the graph.

Only retreat when it runs out of new nodes.

Implementation: Use a stack as an auxiliary data structure. Can also be implemented recursively.

Running Time: $O(m + n)$.

Applications: This could be used for analyzing reachability, linearizability, connectedness.

BFS: The algorithm makes shallow but broad exploration into the graph.

Visit nodes by increasing distances.

Implementation: Could use a queue as an auxiliary data structure.

Running Time: $O(m + n)$

Applications: This could be used for computing distances.

Part II: Distances in Weighted Graphs

Weighted Graphs

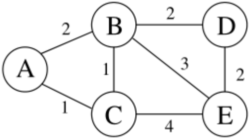
In real appliations, we need to find distances in weighted graphs, that are graph whose edges have integer weights.

Weighted Graphs

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 a graph and $w : E \to \mathbb{Z}$ is a weight function that assigns each edge with an integer weight.



Weighted Graphs

Weighted Graph Representation

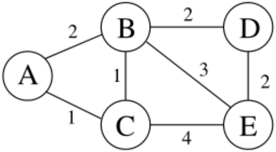
Question: How do we represent a weighted graph?

Weighted Graphs

Weighted Graph Representation

Question: How do we represent a weighted graph?

Answer: We can again have adjacency matrix or adjacency list representations.



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

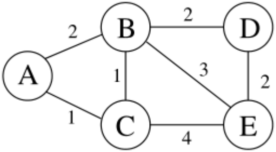
Adjacency Matrix representation of weighted graphs

Weighted Graphs

Weighted Graph Representation

Question: How do we represent a weighted graph?

Answer: We can again have adjacency matrix or adjacency list representations.



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

Question: Can we compute distances in a weighted graph?

Distances in Weighted Graphs

Distances Weighted Graphs

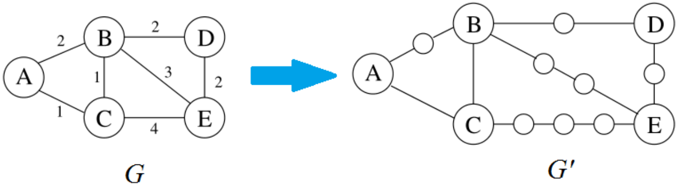
Question: Can we compute distances in a weighted graph?
Answer: Yes!

Distances in Weighted Graphs

Distances Weighted Graphs

Question: Can we compute distances in a weighted graph?
Answer: Yes!

Subdivide edges into a sequence of unit-length edges.



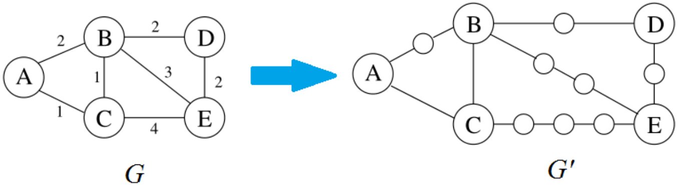
Convert a weighted graph G into an unweighted graph G'

Distances in Weighted Graphs

Distances Weighted Graphs

Question: Can we compute distances in a weighted graph?
Answer: Yes!

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.

Question

How do we compute distances in a weighted graph more efficiently?

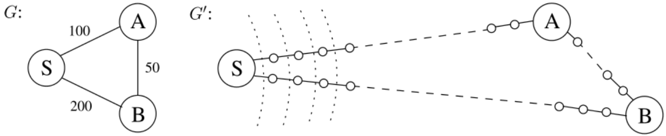
Shortest Path Problem (Single-Sourced)

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

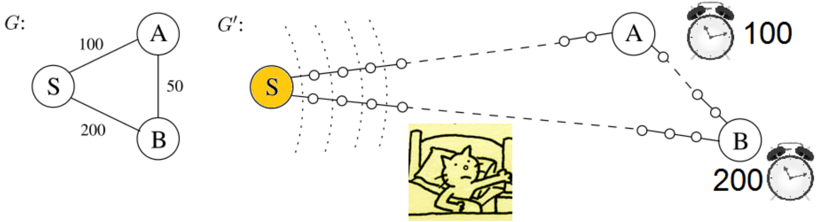
- We can transform a weighted graph into an unweighted one.
- The approach is very inefficient because there could be a large number of auxiliary nodes
- We don't care about the distances on these auxiliary nodes
- We could just go to sleep when BFS visits these auxiliary nodes
- But we need to wake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



Setting Alarm Clocks

Strategy

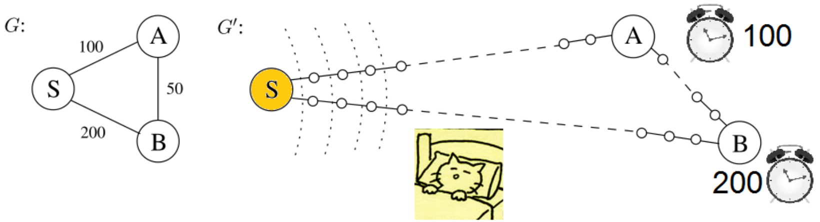
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

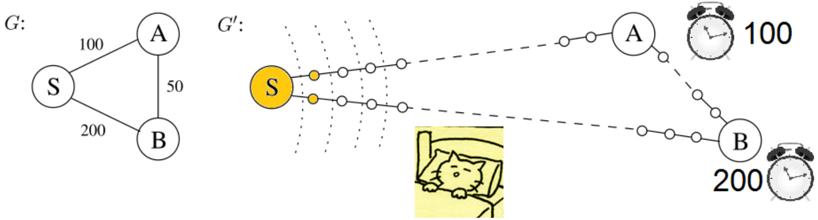
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

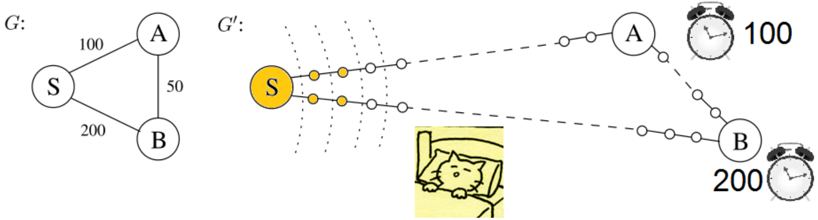
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

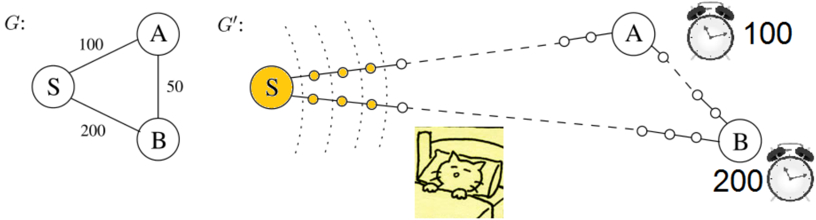
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

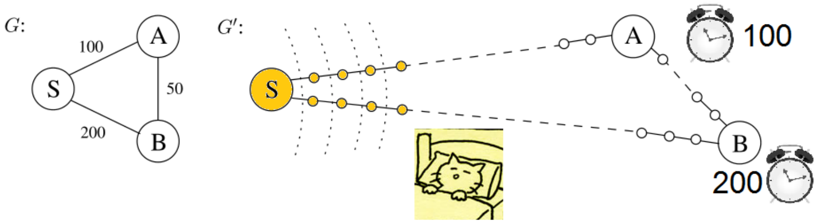
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

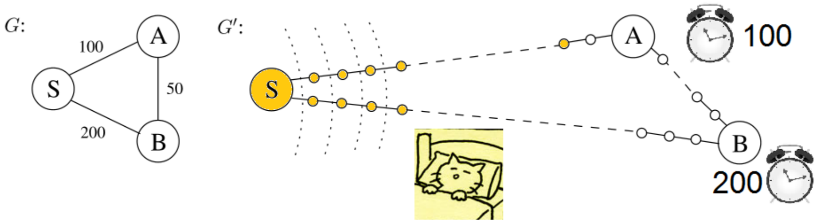
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

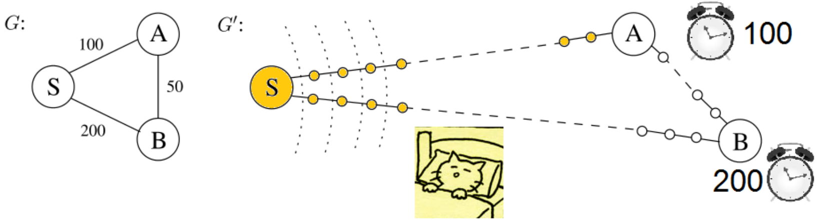
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

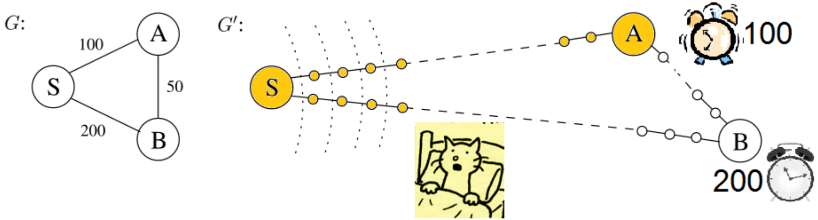
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

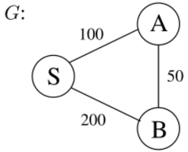
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks

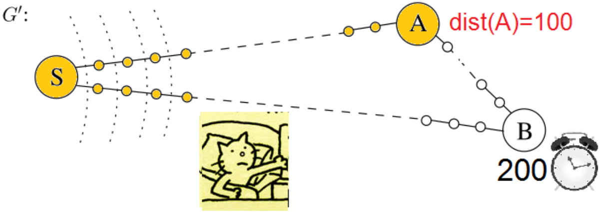


Setting Alarm Clocks

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

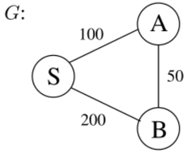


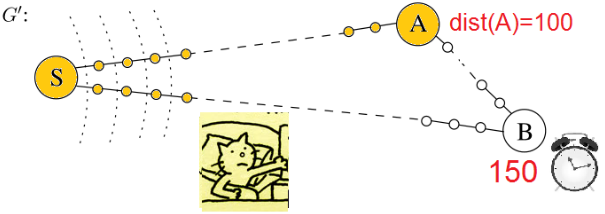


Setting Alarm Clocks

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

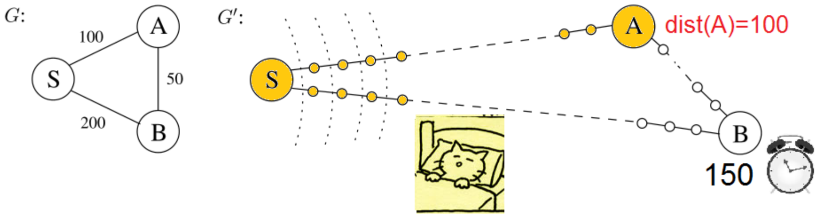




Setting Alarm Clocks

Strategy

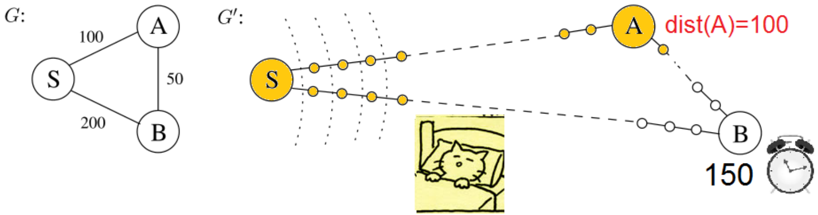
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

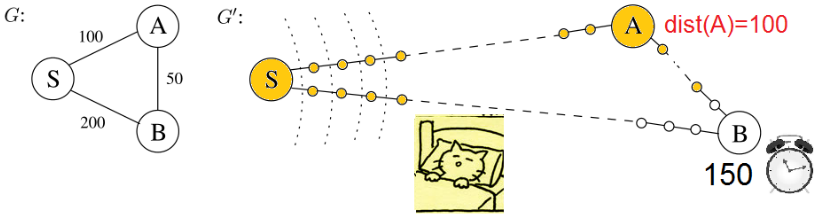
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

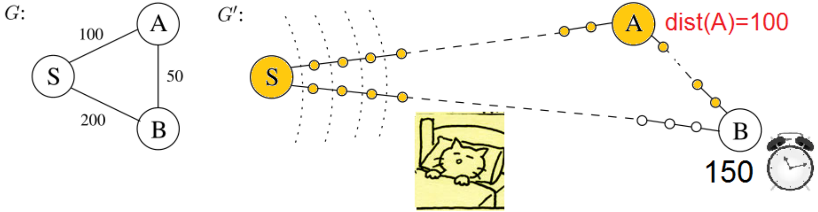
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

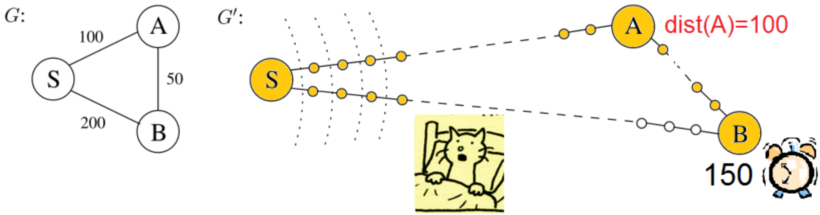
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



Setting Alarm Clocks

Strategy

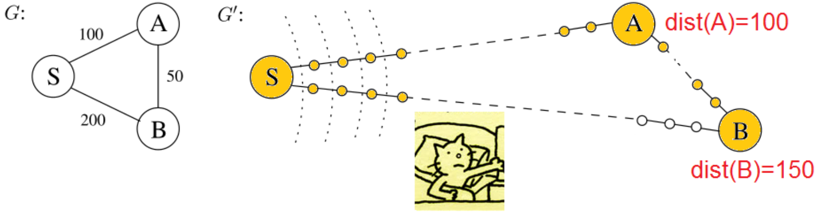
- Maintain an alarm clock for each node. The time we set on an alarm clock is an expected time for visiting this node.
- Whenever an alarm clock goes off, wake up and check which node is reached. Then reset the other clocks



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



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, write $dist(u) = T$
8. Discard this alarm clock
9. For each neighbour 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

"Alarm Clock Algorithm"

Implementing the Alarm Clocks

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

“Alarm Clock Algorithm”

Implementing the Alarm Clocks

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

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

"Alarm Clock Algorithm"

Implementing the Alarm Clocks

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

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

“Alarm Clock Algorithm”

Implementing the Alarm Clocks

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

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

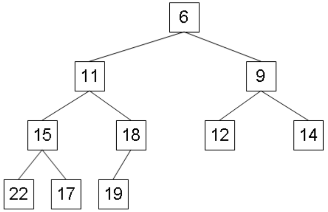
This is a Priority Queue!

Priority Queue

Priority Queues

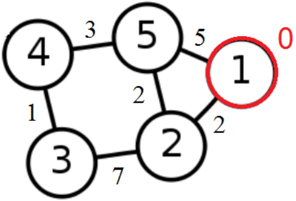
A priority queue is a data structure that store a collection of $(element, key)$ pair where the *key* of an element is an integer value and allows the following operations:

- $Insert(e, k)$: 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

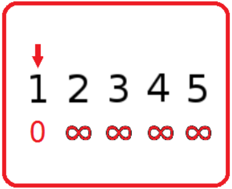


Dijkstra's Algorithm

Implementing the "Alarm Clocks" using Priority Queue:
- Maintain a priority key set for each node
- Maintain a set of confirmed nodes

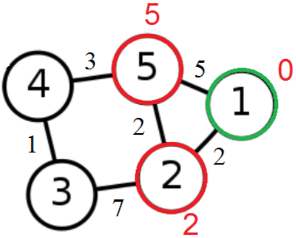


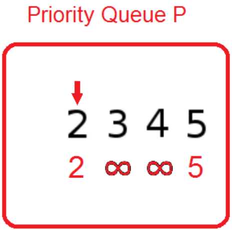
Priority Queue P



Dijkstra’s Algorithm

Implementing the “Alarm Clocks” using Priority Queue:
- Maintain a priority key set for each node
- Maintain a set of confirmed nodes

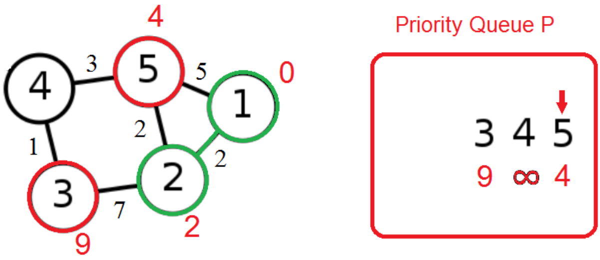




Dijkstra's Algorithm

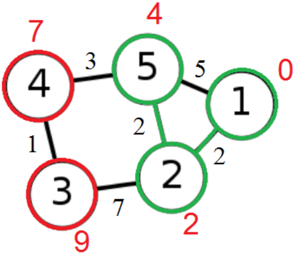
Implementing the "Alarm Clocks" using Priority Queue:

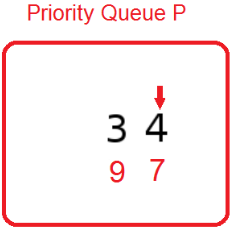
- Maintain a priority key set for each node
- Maintain a set of confirmed nodes



Dijkstra's Algorithm

Implementing the "Alarm Clocks" using Priority Queue:
- Maintain a priority key set for each node
- Maintain a set of confirmed nodes

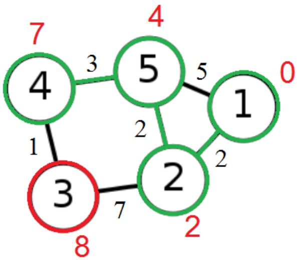


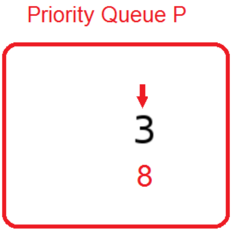


Dijkstra's Algorithm

Implementing the "Alarm Clocks" using Priority Queue:

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

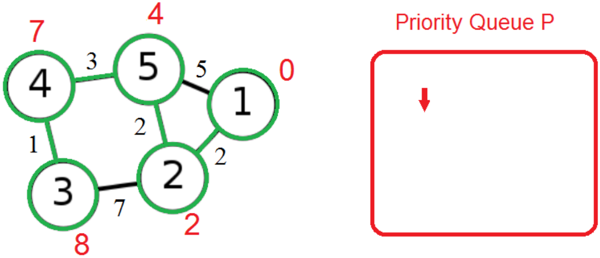




Dijkstra's Algorithm

Implementing the "Alarm Clocks" using Priority Queue:

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



Dijkstra's Algorithm

Algorithm Dijkstra(G, s)

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

$dist(s) \leftarrow 0$
Initialize a set $R \leftarrow \{s\}$
Initialize a priority queue P containing $(s, 0)$
`for` $u \in V, u \neq s$ do
    $dist(u) \leftarrow \infty$
    $prev(u) \leftarrow null$
    $P.Insert(u, \infty)$
`while` P is not empty do
    $u \leftarrow P.DeleteMin()$
    Add u to R
    `for` $(u, v) \in E$ where $v \notin R$ do
        `if` $dist(u) + w(u, v) < dist(v)$ do
            $dist(v) \leftarrow dist(u) + w(u, v)$
            $P.ResetKey(v, dist(v))$
            $prev(v) \leftarrow u$

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

- 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

- 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

- 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.

Dijkstra's Algorithm

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: $O((m+n)\log n)$

Dijkstra's Algorithm

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: $O((m+n)\log n)$

Dijkstra's Algorithm

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., $m < \frac{n^2}{\log n}$

Dijkstra's Algorithm

Different Priority Queues

Which one is better for Dijkstra's Algorithm?

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

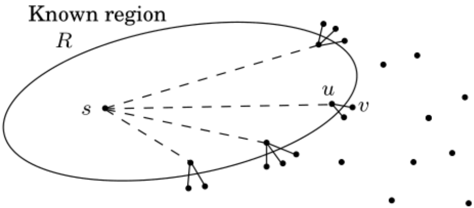
  Preferred when there are a lot of edges, i.e., $m \geq \frac{n^2}{\log n}$
- Binary Heap: $O((m+n)\log n)$

  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



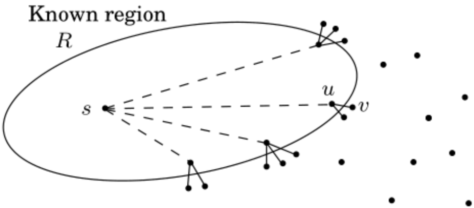
Exploring Graphs

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)

Edsger Dijkstra's Immortality

Edsger Dijkstra 1930 - 2002