

CSU22012: Data Structures and Algorithms II

Lecture 0: Intro and Logistics

Ivana.Dusparic@scss.tcd.ie

Timetables

- › Lectures

- Monday 4-5
 - Thursday 1-2
 - Friday 9-10

- › Labs

- Monday 11-12
 - Thursday 12-1

Lectures – Hilary Term

- › Reading week – March 15th -19th no lectures
- › Easter holidays – April 2nd Friday and 5th Monday no lectures
- › Live but recorded
- › Attendance in lectures – not mandatory but
 - In-class exercises can be submitted and count as a potential *bonus mark* of up to 1% each, up to total of 10%
 - There will be exercises in most lectures, so probably up to 20-ish during the term

Labs – Hilary Term

- › Labs are optional – no new materials will be covered, but can get help with assignments if stuck
- › Groups are pre-assigned per course so please attend your scheduled lab session
- › Assignment issues/questions
 - POST ON BLACKBOARD – no assignment questions will be answered over email – everyone usually has same questions/issues, so let's make the questions/answers of benefit to everyone
 - Before posting, check if question came up before
- › Grades issues/questions
 - Erika Foncesca fonsecae@tcd.ie – TA is the first point of contact for any marks questions - unresolved issues get escalated to me

Course Material

- › Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
- › Lecture notes and assignments will be posted on Blackboard
<https://tcd.blackboard.com/>
- › Assignments
 - Submission **both through Web-CAT and Blackboard**
 - Marking through Web-CAT
<http://webcat.scss.tcd.ie/cs2012/WebObjects/Web-CAT.woa> (only accessible from college network – use VPN to connect from home)
- › In-class exercises
 - Submission **only through Blackboard on the day of the lecture**

Assessment

- › 100% coursework
- › Deadlines
 - No extensions - apart from medical cert or note from tutor – please set reminders
 - Late submissions: mark docked 20% per day
- › Plagiarism – all submissions will be run through Jplag

Assessment Schedule

#	Topic	Issued	Due	Worth
1	Sorting	Week 2	Week 5	20%
2	Graphs	Week 5	Week 8	20%
3	E-test		Week 7	20%
4	<i>Group</i> project – sorting, graphs, strings	Week 8	Week 11	40%
--	In-class exercises	Every-ish lecture	On the day of the lecture	Bonus +10%

E-test

- › Week 7 (after reading week)
- › 1-hour MCQ blackboard test, timed, to be taken within assigned 24-hour window
- › Pick a day within week 7
- › POLL

Assignments

- › Assignments 1 (sorting) and 2 (graphs)
- › Same as last term – automated marking through webcat
- › Group assignment
 - Groups of 4 – you'll be allowed pick your own group
 - Group mark, but individual contributions will be monitored through git commits
 - More details after reading week
- › What day do you want assignments to be due? POLL

In-class exercises

- › You will need a pen and paper (or for simpler ones you can use a text document or a drawing program)
- › Take a pic and upload to blackboard – same day accepted only
- › Only for bonus marks so need to email to ask for exemptions etc
- › 5-minute breakout-group exercises – you will need to submit individual answers but can discuss solutions/approaches within your randomly assigned breakout group
 - Trial run?
- › Not marked for correctness but for effort – half or full mark

So what are we
actually going to
learn?



Course content - Review and expand

- › Sorting algorithms
 - Insertion sort, heapsort ✓
 - Selection sort, shellsort, mergesort, quicksort
 - Space and time trade offs
 - Select and compare based on input type and size
- › Algorithmic approaches
 - Brute force, exhaustive search, decrease and conquer, divide and conquer, greedy, dynamic programming ...

Course content – New Topics

- › Graphs – shortest path
 - Dijkstra
 - Depth-first, breadth-first search, Prim, Kruskal, Topological sort
 - Shortest paths - Bellman-Ford, Floyd-Warshall
 - What to use based on graph – directed, undirected, acyclic, negative edge weights etc
- › Network flow algorithms
 - Maxflow, Ford-Fulkerson
- › Strings
 - String sorts
 - Substring search

Tools

- › Blackboard and Blackboard Collaborate Ultra
 - Lectures
 - Labs
 - in-class polls
 - Assignment submission
 - In-class exercise submission
- › Web-CAT
 - Assignment submission, testing and marking
- › Version control – Git – mandatory for group assignment but highly recommend it for all
 - Github, bitbucket, gitlab
 - gitlab.scss.tcd.ie

Highlights

- › Lectures live but recorded
- › Labs not mandatory, only if you need help with assignment
- › In-class exercises for bonus marks
- › Any non-private questions – Blackboard forum only!

Questions?

CS22012: Data Structures and Algorithms II

Topic 01: Sorting Algorithms

Ivana.Dusparic@scss.tcd.ie

Lecture Outline

- › Introduce algorithm design techniques
- › Review sorting algorithms
 - Insertion sort
- › Learn some new ones
 - Bubble sort
 - Selection sort
 - Shellsort
 - Mergesort
 - Quicksort
- › Analyse and classify by
 - Order of growth
 - Best, average, worst running time
 - Design approach
 - Stable vs unstable
- › Textbook and lecture notes: Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne

Algorithm Design Approaches

Algorithm design

- › Brute-force/exhaustive search
- › Decrease and conquer
- › Divide and conquer
- › Transform and conquer
- › Greedy
- › Dynamic programming

Introduction to the Design and Analysis of Algorithms,
Anany Levitin, 3rd edition, Pearson, 2012

Brute-force/exhaustive search

- › Systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement
- › Often simplest to implement but not very efficient
- › Impractical for all but smallest instances of a problem
- › Examples:
 - Selection sort
 - Bubble sort
 - In graphs – depth-first search (DFS), breadth-first search (BFS)

Decrease and conquer

- › Establish relationship between a problem and a smaller instance of that problem
- › Exploit that relationship top down or bottom up to solve the bigger problem
- › Naturally implemented using recursion
- › Examples
 - Insertion sort
 - In graphs – topological sorting

Divide and conquer

- › Divide a problem into several subproblems of the same type, ideally of the same size
- › Solve subproblems, typically recursively
- › If needed, combine solutions
- › Examples:
 - Mergesort
 - Quicksort
 - Binary tree traversal – preorder, inorder, postorder
 - › Visit root, its left subtree, and its right subtree

Transform and conquer

- › Modify a problem to be more amenable to solution, then solve
 - Transform to a simpler/more convenient instance of the same problem – *instance simplification*
 - Transform to a different representation of the same instance – *representation change*
 - Transform to an instance of a different problem for which an algorithm is available – *problem reduction*
- › Examples:
 - Balanced search trees – AVL trees, 2-3 trees
 - Gaussian elimination – solving a system of linear equations

Dynamic programming

- › Similar to divide and conquer, solves problems by combining the solutions to subproblems
 - In divide and conquer subproblems are disjoint
 - In dynamic programming, subproblems overlap, ie share subsubproblems
 - › Solutions to those are stored, index and reused
- › Examples:
 - A more efficient solution to Knapsack problem
 - Warshall's and Floyd's shortest path algorithms

Greedy

- › Always make the choice that looks best at the moment
 - Does not always yield the most optimal solution, but often does
- › Examples
 - Graphs:
 - › Dijkstra – find the shortest path from the source to the vertex nearest to it, then second nearest etc
 - › Prim
 - › Kruskal
 - Strings
 - › Huffman coding tree

Example: Binary Search

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

- Too small, go left.
 - Too big, go right.
 - Equal, found.



successful search for 33

Binary Search is an example of what kind of algorithm:

- › A – divide and conquer
- › B – decrease and conquer
- › C – brute-force
- › D – greedy

Fun fact: ex-bug in JDK implementation of binary search

› <https://research.googleblog.com/2006/06/extr-extra-read-all-about-it-nearly.html>

- Blog post by Joshua Bloch, Software Engineer who implemented binary search in `java.util.Arrays`
- Undiscovered for 9 years

```
1:  public static int binarySearch(int[] a, int key) {  
2:      int low = 0;  
3:      int high = a.length - 1;  
4:  
5:      while (low <= high) {  
6:          int mid = (low + high) / 2;  
7:          int midVal = a[mid];  
8:  
9:          if (midVal < key)  
10:             low = mid + 1  
11:          else if (midVal > key)  
12:             high = mid - 1;  
13:          else  
14:              return mid; // key found  
15:      }  
16:      return -(low + 1); // key not found.  
17:  }
```

Sorting Algorithms

Sorting

- › Sort data in order
 - Numbers in ascending/descending order
 - Strings alphabetically
 - Dates chronologically
 - etc
- › Total order
 - Ascending $x_0 \leq x_1 \leq x_2 \leq x_3 \leq \dots \leq x_{n-1}$
 - Descending $x_0 \geq x_1 \geq x_2 \geq x_3 \geq \dots \geq x_{n-1}$

Total order

$$x_0 \leq x_1 \leq x_2 \leq x_3 \leq \dots \leq x_{n-1}$$

› Is a binary relation \leq that satisfies

- **Antisymmetry:** if both $v \leq w$ and $w \leq v$, then $v = w$.
- **Transitivity:** if both $v \leq w$ and $w \leq x$, then $v \leq x$.
- **Totality:** either $v \leq w$ or $w \leq v$ or both.

Useful sorting abstractions

Less. Is item v less than w ?

```
private static boolean less(Comparable v, Comparable w)
{  return v.compareTo(w) < 0;  }
```

Exchange. Swap item in array a[] at index i with the one at index j.

```
private static void exch(Comparable[] a, int i, int j)
{
    Comparable swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

- public interface Comparable <T> -This interface imposes a total ordering on the objects of each class that implements it.
- public int compareTo (Item x)
- Implemented by String, Integer, Double, Short, Calendar, Year, etc

Performance Analysis

- › Cost models
 - Running time
 - Memory cost
- › Methods to measure/express
 - Tilde notation, $\tilde{T}(n)$ – counting number of executions of certain operations as a function of input size n
- › Order of growth classification
 - Big Theta $\Theta(n)$ – asymptotic order of growth
 - Big Oh $O(n)$ - upper bound
 - Big Omega $\Omega(n)$ – lower bound

Performance Analysis

- › Time complexity
 - Worst Case Analysis – usually done
 - › Upper bound on running time of an algorithm
 - › Must know the case that causes the maximum number of operations to be performed, eg in linear search, if the element is not in the array
 - Average – not easy to do in practice
 - › Take all possible inputs and calculate computing time for all of the inputs, and average
 - › Must know/predict distribution of cases
 - Best – is it any use if worst case bad?
 - › Lower bound on running time of an algorithm
 - › Must know the case that causes the minimum number of operations to be performed

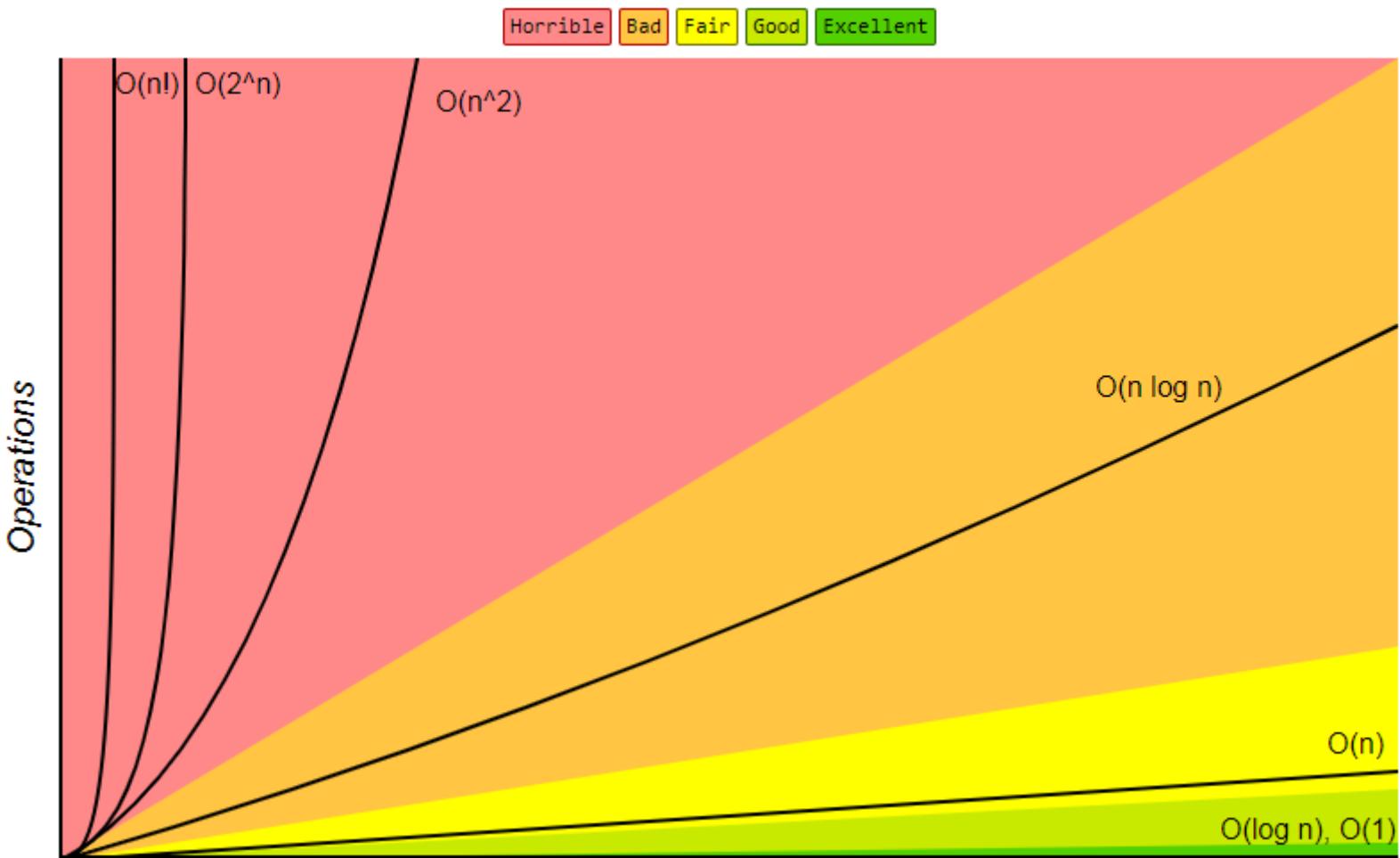
Refresher

- › Refer to semester 1 slides for more details on performance analysis

Common order-of-growth classifications

N^3	cubic	for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Big-O Complexity Chart





WASSIM CHEGHAM

@manekinekko

Follow



We should reconsider big O notations...

$\Theta(1)$ → $\Theta(\smiley)$

$\Theta(\log(n))$ → $\Theta(\smile)$

$\Theta((\log(n))^c)$ → $\Theta(\angel)$

$\Theta(n)$ → $\Theta(\smile)$

$\Theta(n \log(n))$ → $\Theta(\neutral)$

$\Theta(n^2)$ → $\Theta(\worried)$

$\Theta(n^c)$ → $\Theta(\crying)$

$\Theta(c^n)$ → $\Theta(\angry)$

$\Theta(n!)$ → $\Theta(\surprised)$

9:02 AM - 24 Oct 2018

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	
<u>Quicksort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
<u>Mergesort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Timsort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Heapsort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
<u>Bubble Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Insertion Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Selection Sort</u>	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Tree Sort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
<u>Shell Sort</u>	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
<u>Bucket Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
<u>Radix Sort</u>	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
<u>Counting Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
<u>Cubesort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

computer	insertion sort (N^2)			mergesort ($N \log N$)		
	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Visualisation of sorting algorithm performance

- › <https://www.youtube.com/watch?v=ZZuD6iUe3Pc>
- › Tested for different types of input

Why do we need so many then?

- › No Free Lunch Theorem
- › Different applications/different behaviour based on input
- › Examples
 - Merge sort – useful for linked lists
 - Heapsort – sorting arrays, predictable, very little extra RAM
 - Quicksort – excellent average-case behaviour
 - Insertion sort – good if your list is already almost sorted
 - Bubble sort – if small enough data set, it is the simplest to implement
- › Also, a handy way to learn different algorithm design strategies on the same example!

Why do we need so many sorting algorithms?

- › Bubble sort uses (from exam answers)
- › What kind of problems is bubble sort useful for?
 - “it is useful for getting points on the exam”
 - “it is useful if you want to sort a list really slowly”
 - “it is useful for when a teacher in school asks you to line up by height”

Stability of Sorting Algorithms

- › Stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted
- › Do we care?
 - NO: When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key
 - NO: If all keys are different.
 - YES: if duplicate keys and want to maintain original order by eg secondary key.
 - When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. Stability is also not an issue if all keys are different.

Stability of Sorting Algorithms

sorted by time	sorted by location (not stable)	sorted by location (stable)
Chicago 09:00:00	Chicago 09:25:52	Chicago 09:00:00
Phoenix 09:00:03	Chicago 09:03:13	Chicago 09:00:59
Houston 09:00:13	Chicago 09:21:05	Chicago 09:03:13
Chicago 09:00:59	Chicago 09:19:46	Chicago 09:19:32
Houston 09:01:10	Chicago 09:19:32	Chicago 09:19:46
Chicago 09:03:13	Chicago 09:00:00	Chicago 09:21:05
Seattle 09:10:11	Chicago 09:35:21	Chicago 09:25:52
Seattle 09:10:25	Chicago 09:00:59	Chicago 09:35:21
Phoenix 09:14:25	Houston 09:01:10	Houston 09:00:13
Chicago 09:19:32	Houston 09:00:13	Houston 09:01:10
Chicago 09:19:46	Phoenix 09:37:44	Phoenix 09:00:03
Chicago 09:21:05	Phoenix 09:00:03	Phoenix 09:14:25
Seattle 09:22:43	Phoenix 09:14:25	Phoenix 09:37:44
Seattle 09:22:54	Seattle 09:10:25	Seattle 09:10:11
Chicago 09:25:52	Seattle 09:36:14	Seattle 09:10:25
Chicago 09:35:21	Seattle 09:22:43	Seattle 09:22:43
Seattle 09:36:14	Seattle 09:10:11	Seattle 09:22:54
Phoenix 09:37:44	Seattle 09:22:54	Seattle 09:36:14

Stability when sorting on a second key

- › Stable sorting algorithms: Insertion sort, bubble sort, merge sort

Memory requirements/In-place algorithms

- › Transforms input without additional auxiliary data structure, eg array
- › A small amount of extra storage space is allowed for auxiliary variables
- › The input is usually overwritten by the output as the algorithm executes
- › In-place algorithm updates input sequence only through replacement or swapping of elements

- › Affects space complexity of an algorithm
- › Selection, insertion, shell, quick

Checkpoint – is
the material clear
enough?



Which of these O(n) has the highest complexity (worst worst performance)?

- › A – $O(\log n)$
- › B – $O(n \log (n))$
- › C – $O(n^2)$
- › D – $O(2^n)$

Finally: let's do actual sorting algorithms

Review: Insertion Sort

Insertion sort

› Algorithm

1. Start from 1st element of the array
2. Shift element back until you find a smaller element – maintain the array from 0 to (current position) sorted.
3. Continue to next element
4. Repeat (2) and (3) until the end of the array

Insertion sort

i=1

62 **83** 18 53 07 17 95 86 42 69 25 28

i=2

62 83 **18** 53 07 17 95 86 42 69 25 28

Insertion sort

i=1

62 83 18 53 07 17 95 86 42 69 25 28

i=2

18 **62** 83 53 07 17 95 86 42 69 25 28

i=3

18 62 83 **53** 07 17 95 86 42 69 25 28

Insertion sort

i=1

62 83 18 53 07 17 95 86 42 69 25 28

i=2

18 62 83 53 07 17 95 86 42 69 25 28

i=3

18 53 62 83 07 17 95 86 42 69 25 28

i=4

18 53 62 83 07 17 95 86 42 69 25 28

...

Insertion sort implementation – ints in Java

```
public static int[] insertionSort(int[] input){  
  
    int temp;  
    for (int i = 1; i < input.length; i++) {  
        for(int j = i ; j > 0 ; j--){  
            if(input[j] < input[j-1]){  
                temp = input[j];  
                input[j] = input[j-1];  
                input[j-1] = temp;  
            }  
        }  
    }  
    return input;  
}
```

Insertion sort implementation – ints in Java

```
void insertionSort(int numbers[])
{
    int i, j, index;
    int size = numbers.size;

    for (i = 1; i < size; i++)
    {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j - 1] > index))
        {
            numbers[j] = numbers[j - 1];
            j = j - 1;
        }
        numbers[j] = index;
    }
}
```

Insertion sort exercise

(c) Consider the code for Insertion sort given below.

```
public void iterInsertSort(Comparable[] list) {  
    int N = list.length;  
    for(int i=1; i<N; i++) {  
        for(int j=i; j>0; j--) {  
            if(list[j].compareTo(list[j-1]) < 0) {  
                Comparable temp = list[j];  
                list[j] = list[j-1];  
                list[j-1] = temp;  
            }  
        }  
    }  
}
```

Assume you are given an array containing the following integers 5, 4, 2, 5, 1.

Provide the trace of the array content at each time step of Insertion sort algorithm

Insertion sort exercise

Insertion sort properties

- › Performance
 - Best case?
 - Worst case?
- › O?
- › Stable?
- › In-place?



Insertion sort properties

- › Performance
 - Best case – array is sorted
 - Worst case – array is sorted in reverse order
- › O?
 - Comparisons and swaps (lecture 3.2 in semester 1)
- › But best case performance is $\Omega(n)$
- › Stable? YES
- › In-place? YES
- › Example use: often used to speed up other algorithms like quicksort: you quicksort until the partitions are around 8 items in size and then insertion sort the whole array. This tends to be faster than just allowing quicksort to complete down to one-item partitions.

Bubble Sort

Bubble sort

- › Make multiple passes through a list
- › In each pass, compare adjacent items and exchange those that are out of order
- › Each pass through the list places the next largest value in its proper place

Bubble sort – int array in Java

```
public static void bubbleSort(int[] numArray) {  
  
    int n = numArray.length;  
    int temp = 0;  
  
    for (int i = 0; i < n; i++) {  
        for (int j = 1; j < (n - i); j++) {  
  
            if (numArray[j - 1] > numArray[j]) {  
                temp = numArray[j - 1];  
                numArray[j - 1] = numArray[j];  
                numArray[j] = temp;  
            }  
        }  
    }  
}
```

Bubble Sort

62.0 , 83.0 , 18.0 , 53.0 , 7.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,
swapping 83.0 and 18.0

62.0 , 18.0 , 83.0 , 53.0 , 7.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,
swapping 83.0 and 53.0

62.0 , 18.0 , 53.0 , 83.0 , 7.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,
swapping 83.0 and 7.0

62.0 , 18.0 , 53.0 , 7.0 , 83.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,
swapping 83.0 and 17.0

...

Bubble sort properties

- › Performance
 - Best case?
 - Worst case?
- › O?
- › Stable?
- › In-place?



Bubble sort properties

- › Performance
 - Best case – array is sorted
 - Worst case – array is sorted in reverse order
- › O?
 - Two nested loops $O(n^2)$
- › But best case performance is $\Omega(n)$
- › Stable? YES
- › In-place? YES
- › Question: So how/why is it worse than insertion sort?

Bubble sort properties

- › Simple to implement so easy for small lists
- › If there are no swaps during the pass, means the array is sorted -> can stop
 - Keep track by adding swapNeed=true statement
 - Useful in nearly ordered lists where there are very few passes
- › Bubble sort uses (from exam answers)
- › What kind of problems is bubble sort useful for?
 - “it is useful for getting points on the exam”
 - “it is useful if you want to sort a list really slowly”
 - “it is useful for when a teacher in school asks you to line up by height”

Selection Sort

Selection sort

- › In each iteration find the smallest remaining entry
- › Swap current entry and the one you find

Algorithm. ↑ scans from left to right.

Invariants.

- Entries to the left of ↑ (including ↑) fixed and in ascending order.
- No entry to right of ↑ is smaller than any entry to the left of ↑.



Selection sort

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```



- Identify index of minimum entry on right.

```
int min = i;
for (int j = i+1; j < N; j++)
    if (less(a[j], a[min]))
        min = j;
```



- Exchange into position.

```
exch(a, i, min);
```



Selection sort – int array in Java

```
void sort(int arr[])
{
    int n = arr.length;

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Selection Sort

62.0 , 83.0 , 18.0 , 53.0 , 7.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,

swapping 7.0 and 62.0

7.0 , 83.0 , 18.0 , 53.0 , 62.0 , 17.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,

swapping 17.0 and 83.0

7.0 , 17.0 , 18.0 , 53.0 , 62.0 , 83.0 , 95.0 , 86.0 , 42.0 , 69.0 , 25.0 , 28.0 ,

swapping 18.0 and 18.0

Selection Sort

Sort the digits of your student ID using selection sort, showing intermediate steps

Selection sort properties

- › Performance
 - Best case?
 - Worst case?
- › O?
- › Stable?
- › In-place?



Selection sort properties

- › Performance
- › Worst case?
 - Two nested loops $O(n^2)$
- › Best case performance also $\Omega(n^2)$
 - Insensitive to input – finding the smallest one in one pass, implies nothing about where smallest one will be at the next, so still need a full pass
- › Stable? NO
 - Eg 4 2 3 4 1 – find smallest which is 1, swap with 1st 4 = 1 2 3 4 4
- › In-place? YES
- › Example use
 - Minimal number of swaps/writes (n writes), if want to avoid writing to memory
 - Fast on small input sizes, 20-30

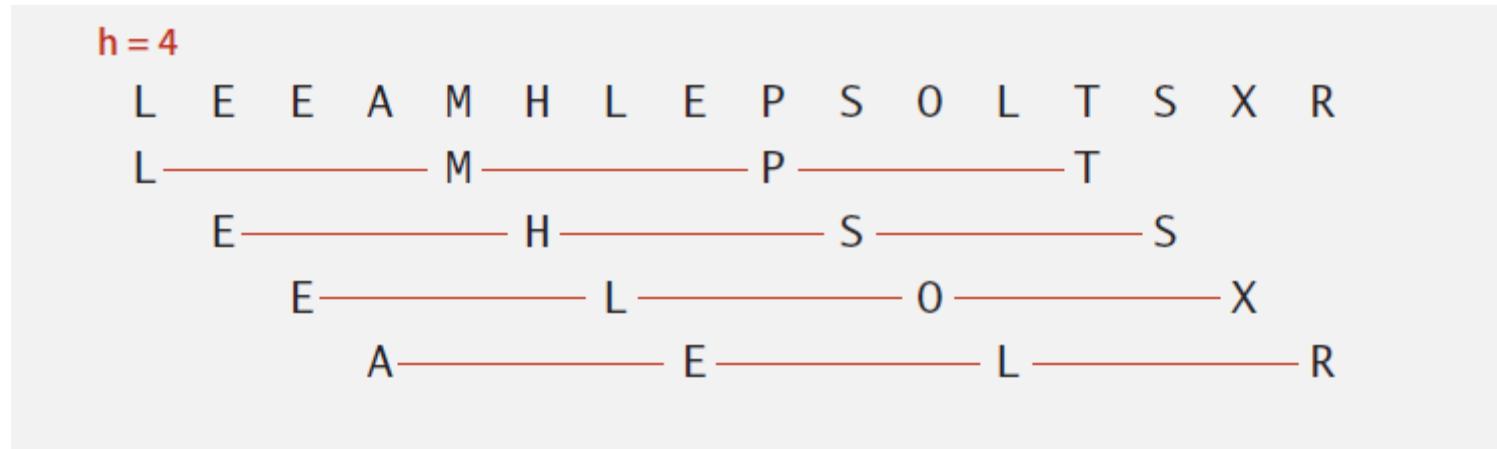
Shellsort

Shellsort

- › Based on Insertion sort
 - Insertion slow for larger lists - it considers only adjacent items, so items move through array only 1 slot at a time
- › Shellsort allows exchanges of entries that are far apart to produce partially sorted arrays, which are then sorted by insertion sort

Shellsort

- › H-sorted array: take every h-th entry (starting anywhere) to get a sorted sequence



- › h independent sorted sequences, interleaved together

Shellsort

- › Use increment sequence of h , ending at $h=1$, to produce a sorted array

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

Shellsort

- › How to select increment sequence?
- › No provably best sequence has been found
- › $\frac{1}{2} (3^k - 1)$
 - Easy to compute and use
 - Performs nearly as well as more sophisticated ones

```
h=1;  
while (h < n/3)  
    h = 3h +1;  
    h-h/3;  
    //1, 4, 13, 40, 121, etc
```

Shellsort – java ints

```
public static void sort(int[] a)    {  
    int N = a.length;  
  
    int h = 1;  
    while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, ... ← 3x+1 increment sequence  
  
    while (h >= 1)  
    { // h-sort the array.  
        for (int i = h; i < N; i++) ← insertion sort  
        {  
            for (int j = i; j >= h && (a[j] < a[j-h]); j -= h)  
                exch(a, j, j-h);  
        }  
  
        h = h/3; ← move to next increment  
    }  
}
```

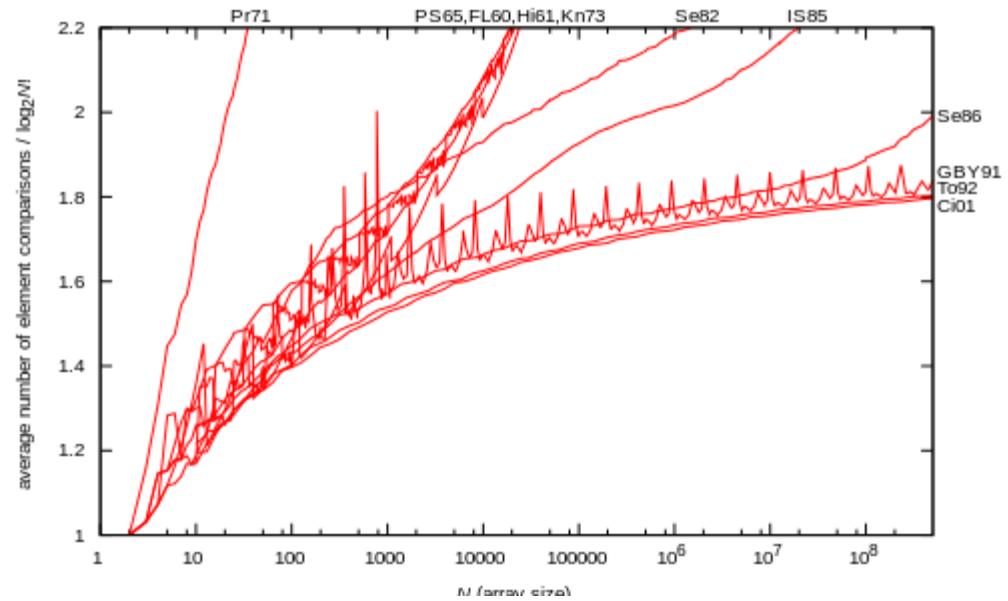
Swap method

```
private static void exch(int[] a, int i, int j){  
    int swap = a[i];  
    a[i] = a[j];  
    a[j] = swap;  
}
```

Shellsort properties

- › <https://en.wikipedia.org/wiki/Shellsort> - per increment formula
- › No precise model
- › $N^{3/2}$, $N^{4/3}$, $N \log N^2$

- › Stable? no
- › In-place? yes



CSU22012: Data Structures and Algorithms II

Merge sort and quick sort

Ivana.Dusparic@scss.tcd.ie

Divide and Conquer Sorting

- › Divide problem into smaller parts
- › Independently solve the parts
- › Combine these solutions to get overall solution
- › 2 common approaches:
 - Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → Mergesort
 - Partition array into items that are “small” and items that are “large”, then recursively sort the two sets → Quicksort

Merge vs quick

- › In Java, `Arrays.sort()` uses **QuickSort** for sorting primitives and **MergeSort** for sorting Arrays of Objects.
 - Why does it matter for Objects and not for primitive data types?

Mergesort

Merge sort

- › Top down merge sort
 - Recursive
 - Divide array in 2 halves, sort each array recursively, merge the arrays
- › Bottom up merge sort
 - Iterative
 - Iterate through array merging subarrays of size 1, size 2, 4, 8, etc

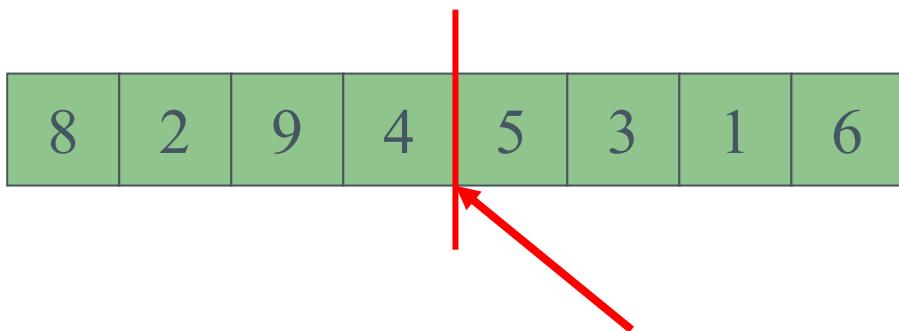
Top down merge sort

a[]															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	S	T	E	X	A	M	P	L
E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
A	E	E	E	E	E	G	L	M	M	O	P	R	R	S	T

Bottom up merge sort

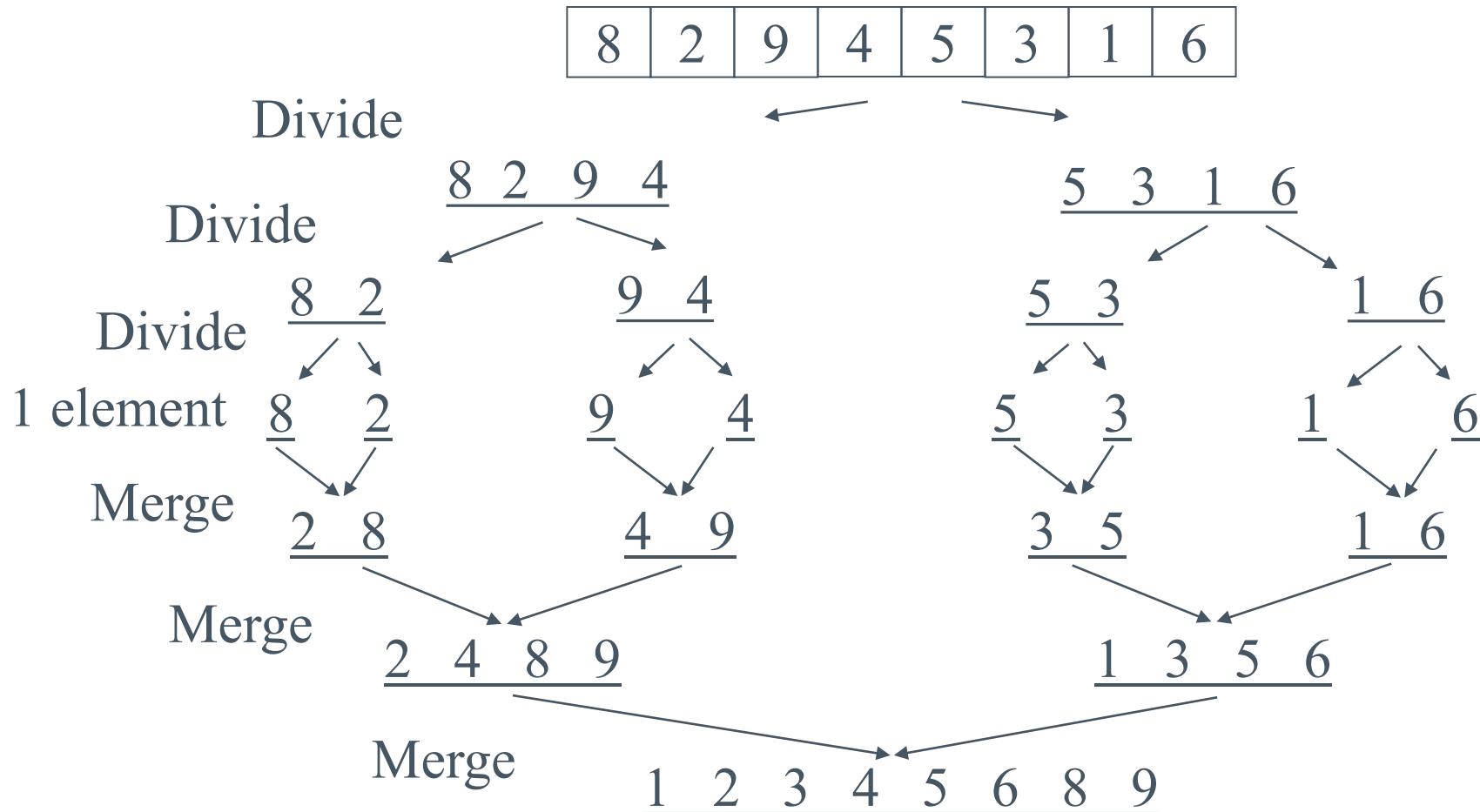
a[i]																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E	
E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E	
E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E	
E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L	
E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L	
E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L	
E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L	
E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P	
E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P	
E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X	
A	E	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Top down merge sort



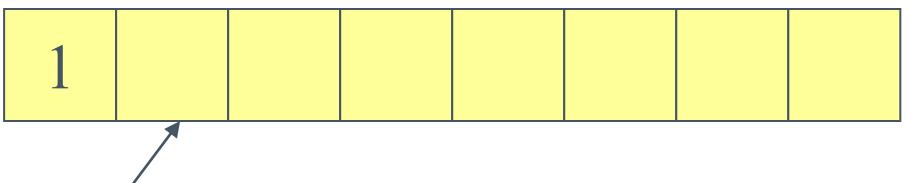
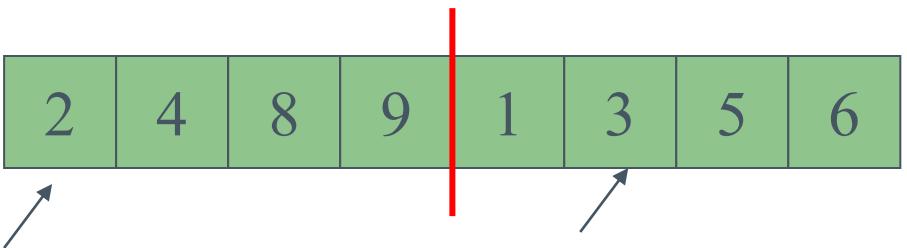
- › Divide it in two at the midpoint
- › Conquer each side in turn (by recursively sorting)
- › Merge two halves together

Top down merge sort



Top down merge sort

- › The merging requires an auxiliary array
 - Requires extra space



Auxiliary array

Top down merge sort Java implementation

- › What methods do we need?

- › public method that passes in array to be sorted

```
public static void sort (Comparable [] a)
```

- › Recursive method with original and auxiliary arrays, and indices of the subarray to be sorted

```
private static void sort (Comparable [] a, Comparable [] aux, int lo,  
int hi)
```

- › Merge method, to merge sorted subarrays, with the 2 arrays to be merged, lowest, highest and midpoint indices

```
private static void merge (Comparable [] a, Comparable [] aux, int lo,  
int mid, int hi)
```

Top down merge sort

- › Create an auxiliary array of the same size as the original one
- › Kick off recursion by passing in 0 and array length-1 as indices (ie the full original array)

```
public static void sort(Comparable[] a)
{
    Comparable[] aux = new Comparable[a.length];
    sort(a, aux, 0, a.length - 1);
}
```

Top down merge sort

- › Recursive method
 - Repeat until lo and hi are equal, ie get to array of length 1
 - Note: $\text{mid} = \text{lo} + (\text{hi}-\text{lo})/2$ to avoid integer overflow

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

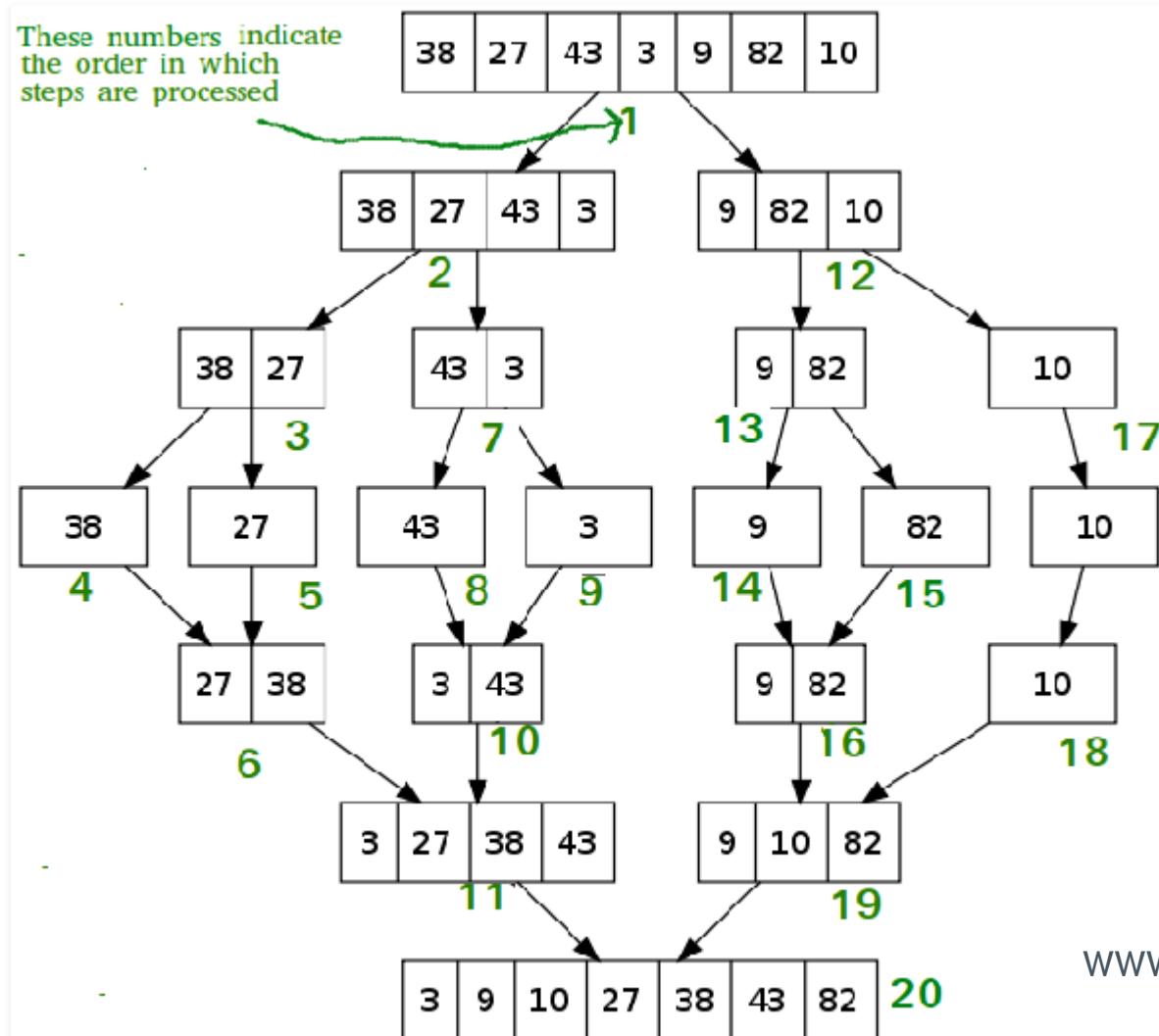
Top down merge sort

- › Merge method
- › Copy the original array into auxiliary one, and then merge elements back into the original one in sorted order

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k]; copy

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if      (i > mid)                  a[k] = aux[j++]; merge
        else if (j > hi)                  a[k] = aux[i++];
        else if (less(aux[j], aux[i]))    a[k] = aux[j++];
        else                            a[k] = aux[i++];
    }
}
```

Top down merge sort



Top down merge sort

	a[]																
lo	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
merge(a, aux, 0, 0, 1)		M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)		E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 1, 3)		E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)		E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)		E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 5, 7)		E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 0, 3, 7)		E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)		E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)		E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, aux, 8, 9, 11)		E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 12, 12, 13)		E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 14, 14, 15)		E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)		E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)		E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, aux, 0, 7, 15)		A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Merge sort exercise

Illustrate merge sort by showing table partitioning and numbering the order in which steps happen (like in the lecture notes example)

7	8	2	4	5	4	3	1
---	---	---	---	---	---	---	---

Merge sort exercise

Illustrate merge sort by showing table partitioning and numbering the order in which steps happen (like in the lecture notes example)

7	8	2	4	5	4	3	1
---	---	---	---	---	---	---	---

Merge sort running time

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

computer	insertion sort (N^2)			mergesort ($N \log N$)		
	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

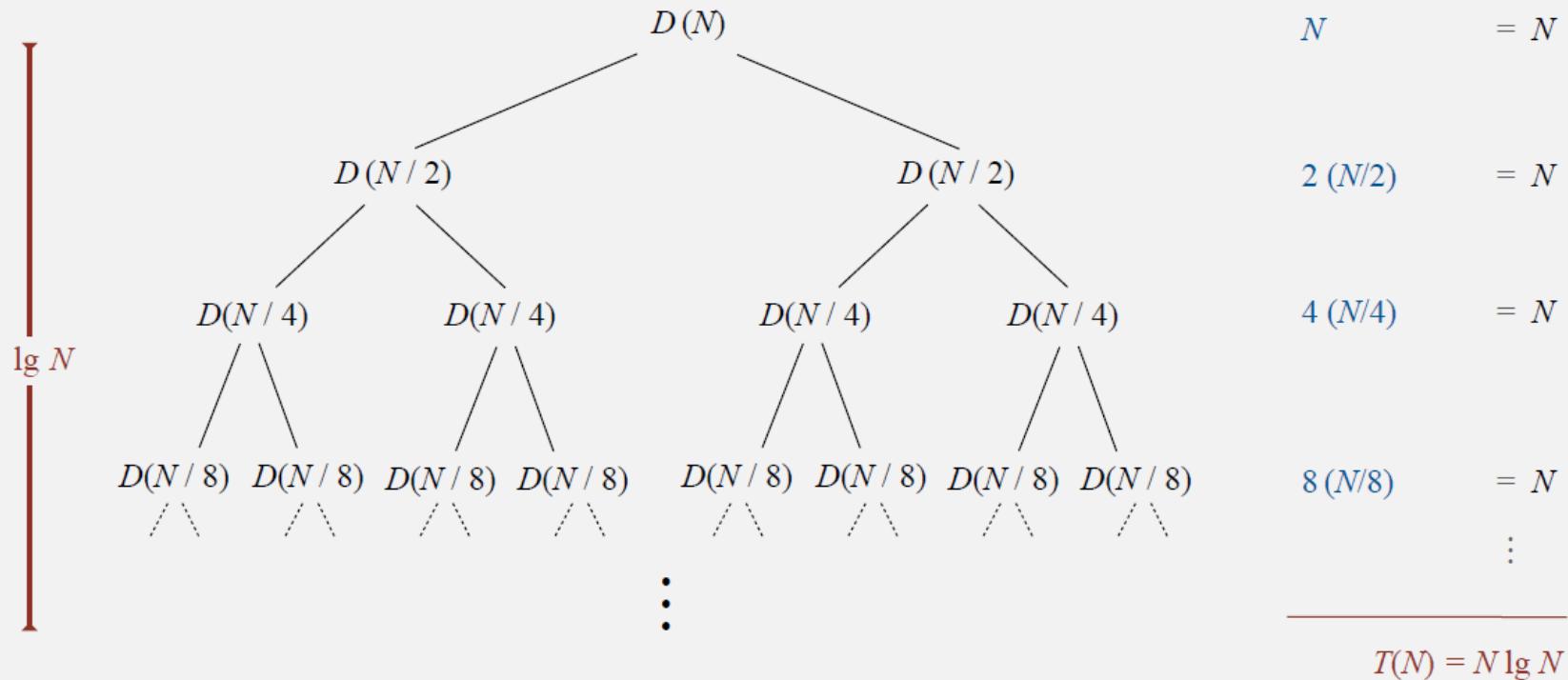
Merge sort

- › Number of compares $< N \lg N$
 - Linearithmic
 - Both average and worst
 - Stable – use “less than” - favours left hand value to right hand one even when they’re equal
- › Number of array accesses $< 6 N \lg N$
- › Memory use – auxiliary array of size N
- › Proofs in Sedgewick

Merge sort

Proposition. If $D(N)$ satisfies $D(N) = 2 D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 1. [assuming N is a power of 2]



Merge sort

Key point. Any algorithm with the following structure takes $N \log N$ time:

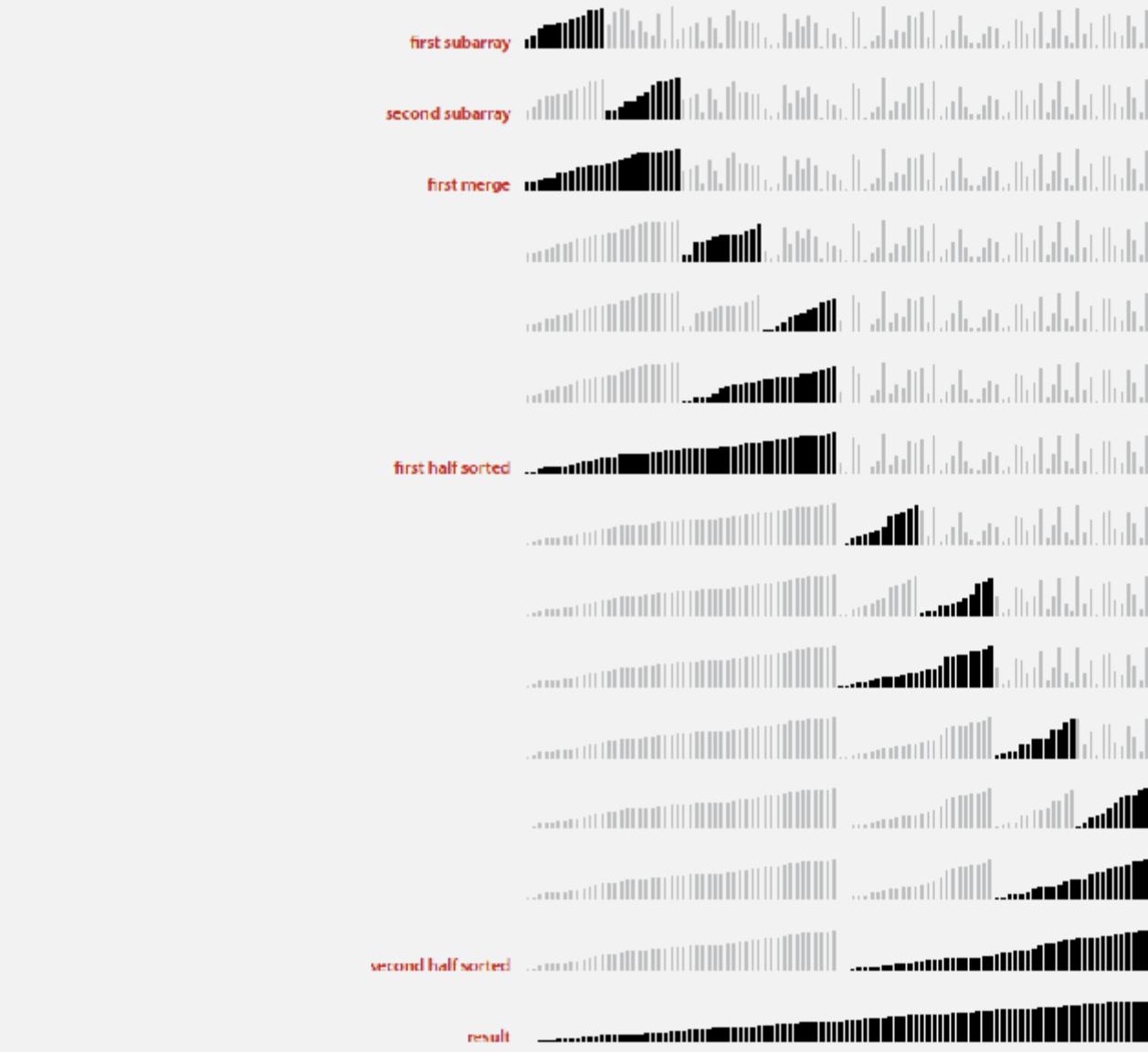
```
public static void linearithmic(int N)
{
    if (N == 0) return;
    linearithmic(N/2); ← solve two problems
    linearithmic(N/2); ← of half the size
    linear(N); ← do a linear amount of work
}
```

Merge sort improvement

- › Too much overhead for small subarrays
- › Cut off to insertion sort for ~10 items

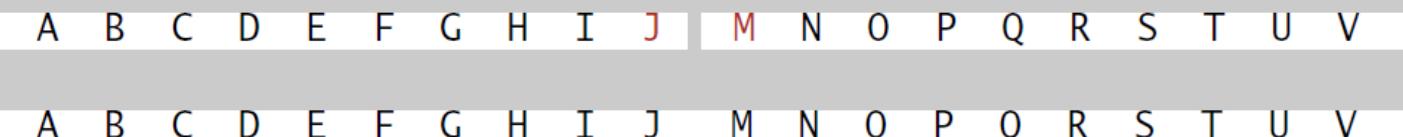
```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1)
    {
        Insertion.sort(a, lo, hi);
        return;
    }
    int mid = lo + (hi - lo) / 2;
    sort (a, aux, lo, mid);
    sort (a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

Mergesort with cutoff to insertion sort: visualization



Merge sort further improvements

- › Stop if already sorted
 - Is largest item in first half smaller than smallest in second half



```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    if (!less(a[mid+1], a[mid])) return;
    merge(a, aux, lo, mid, hi);
}
```

Merge sort further improvements

- › Eliminate the time (but not the space) taken to copy to the auxiliary array used for merging
- › Use two invocations of the sort method
 - one that takes its input from the given array and puts the sorted output in the auxiliary array
 - the other takes its input from the auxiliary array and puts the sorted output in the given array.

Merge sort further improvements

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if (i > mid)          aux[k] = a[j++];
        else if (j > hi)      aux[k] = a[i++];
        else if (less(a[j], a[i])) aux[k] = a[j++]; ← merge from a[] to aux[]
        else                  aux[k] = a[i++];
    }
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort (aux, a, lo, mid);
    sort (aux, a, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

switch roles of aux[] and a[]

assumes aux[] is initialize to a[] once,
before recursive calls

Bottom up mergesort

Merge sort bottom up

- › Pass through array merging subarrays of size 1, 2, 4, etc

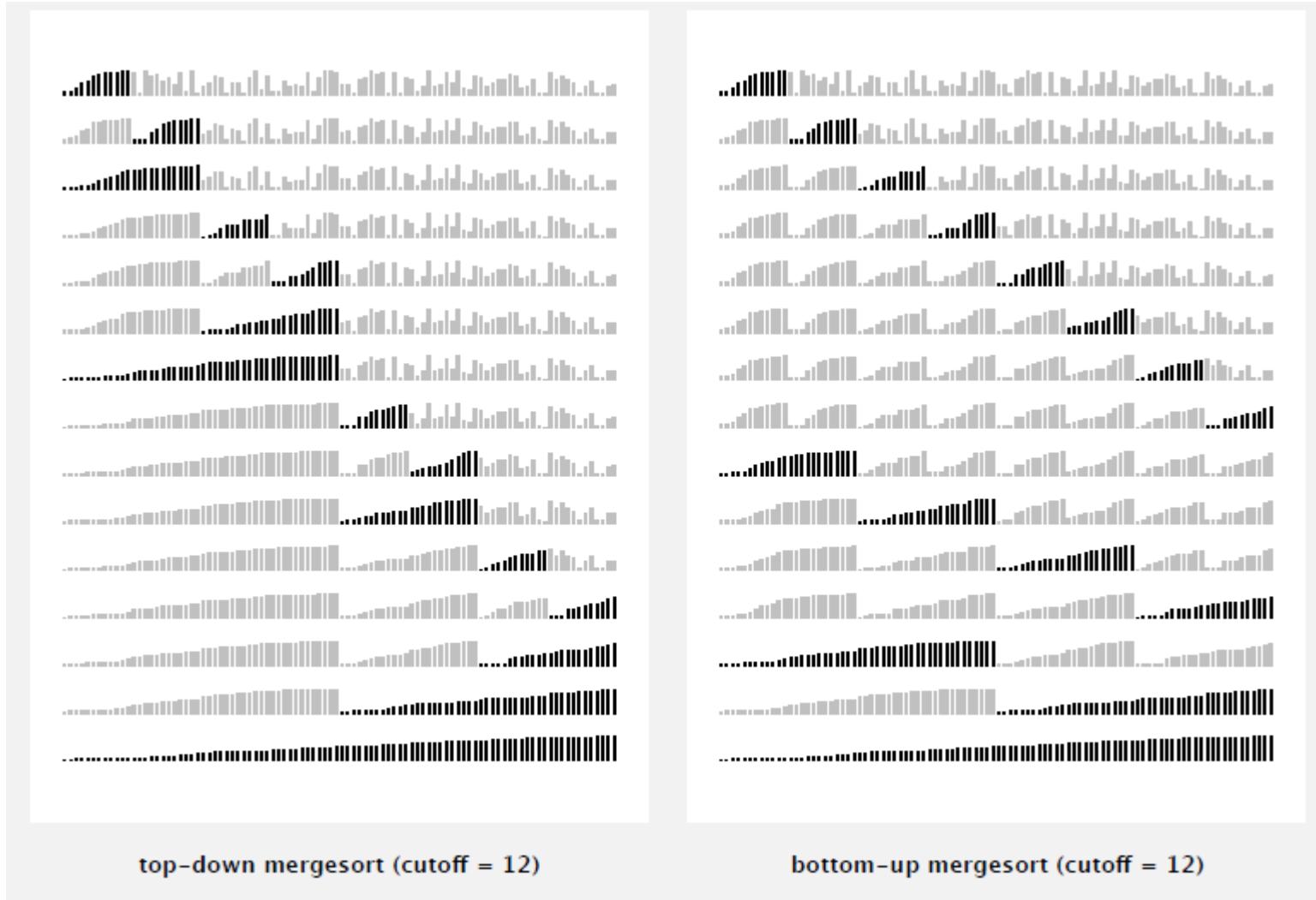
```
public class MergeBU
{
    private static void merge(...)
    { /* as before */ }

    public static void sort(Comparable[] a)
    {
        int N = a.length;
        Comparable[] aux = new Comparable[N];
        for (int sz = 1; sz < N; sz = sz+sz)
            for (int lo = 0; lo < N-sz; lo += sz+sz)
                merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
    }
}
```

Merge sort bottom up

	a[i]																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
sz = 1	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, aux, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, aux, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, aux, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, aux, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E	
merge(a, aux, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E	
merge(a, aux, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E	
merge(a, aux, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L	
sz = 2																	
merge(a, aux, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L	
merge(a, aux, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L	
merge(a, aux, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L	
merge(a, aux, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P	
sz = 4																	
merge(a, aux, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P	
merge(a, aux, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X	
sz = 8																	
merge(a, aux, 0, 7, 15)	A	E	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Merge sort top down vs bottom up



Timsort

- › adaptive sort, combination of
 - natural merge sort - exploit pre-existing order by identifying naturally occurring non-descending sequences (so ascending or equal) - Look for at least 2 elements
 - Use insertion sort to make initial runs
- › Java 7 (for non primitive data types), Python, Android

Quicksort

Quicksort

- › One of top 10 algorithms of 20th century in science and engineering
 - “the greatest influence on the development and practice of science and engineering in the 20th century”
 - <https://www.computer.org/csdl/mags/cs/2000/01/c1022.html>
 - “one of the best practical sorting algorithm for general inputs”
 - inspiration for developing general algorithm techniques for various applications

Quicksort

- › Invented by Tony Hoare in 1959
 - Visiting student in Russia, needed to sort the words before looking them up in dictionary
 - Insert sort was too slow so he developed quicksort, but couldn't implement it until learnt ALGOL and its ability to do recursion
- › Further improvements
 - Sedgwick, Bentley, Yaroslavskiy
 - Dual-pivot implementation in 2009, now implemented in Java 7 onwards

Quicksort

1. Shuffle the array $a[]$ (we'll talk later why)
2. Partition the array so that, for some j
 - $a[j]$ is in place (called pivot)
 - There is nothing larger than $a[j]$ to the left of it
 - There is nothing smaller to the right of it
(where does equal go?)
3. Sort each subarray recursively

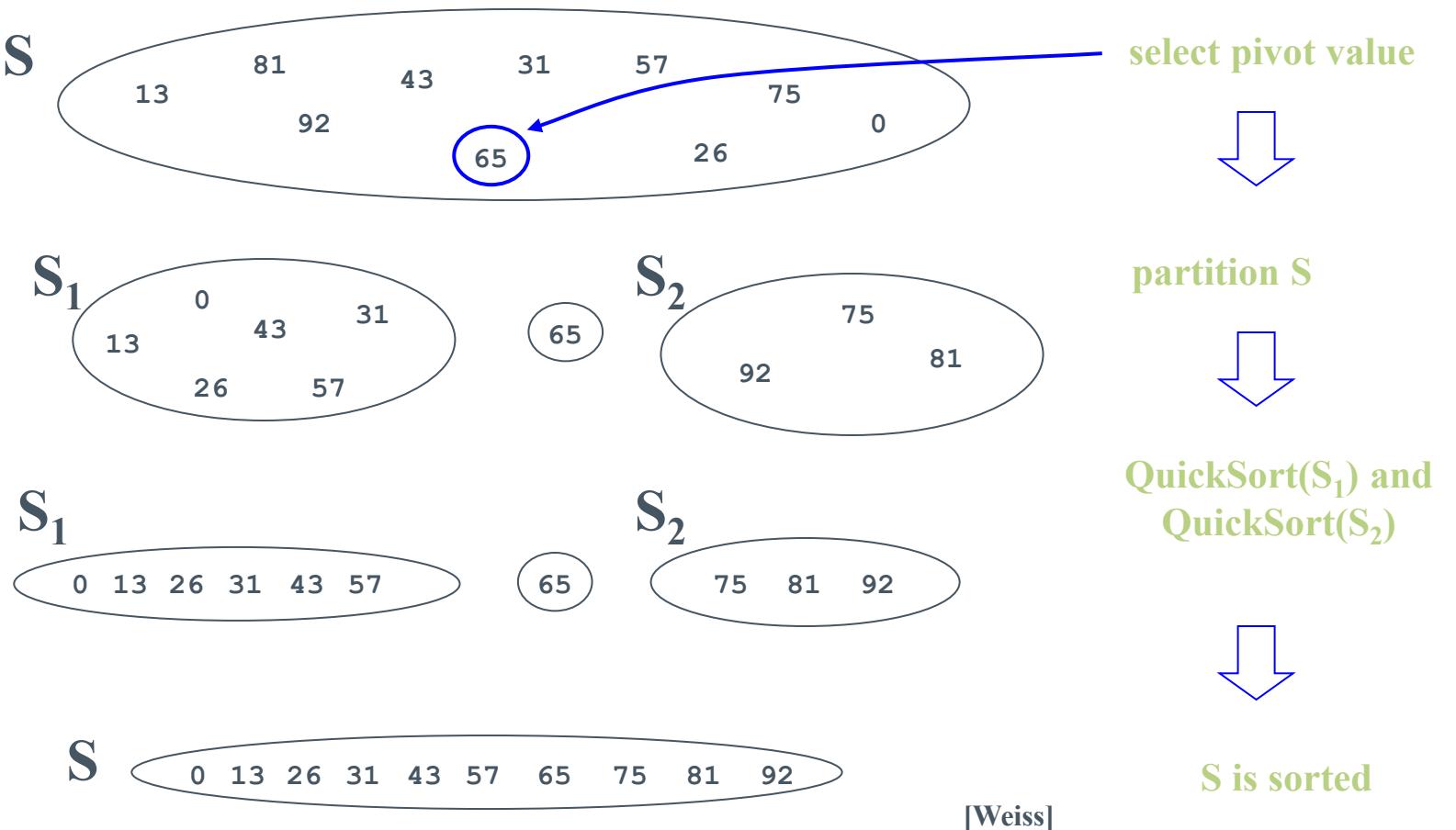
Quicksort

- › To sort an array S
 1. If the number of elements in S is 0 or 1, then return. The array is sorted.
 2. Pick an element v in S . This is the *pivot* value.
 3. Partition $S - \{v\}$ into two disjoint subsets, $S_1 = \{\text{all values } x \leq v\}$, and $S_2 = \{\text{all values } x \geq v\}$.
 4. Return QuickSort(S_1), v , QuickSort(S_2)

Quicksort example

input	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
shuffle	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	<i>not greater</i>					K	<i>not less</i>									
sort left	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
sort right	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort example



Quicksort - details

- › Implement partitioning
 - › recursive
- › Pick a pivot
 - › want a value that will cause $|S_1|$ and $|S_2|$ to be non-zero, and close to equal in size if possible
- Dealing with cases where the element equals the pivot

Quicksort – partitioning

- › Need to partition the array into left and right sub-arrays
 - the elements in left sub-array are \leq pivot
 - elements in right sub-array are \geq pivot
- › How do the elements get to the correct partition?
 - Choose an element from the array as the pivot
 - Make one pass through the rest of the array and swap as needed to put elements in partitions

Quicksort – picking a pivot

- Ideally median value
 - › Expensive, calculating median
 - › Approximate: choose a median of first, middle and last values
- Choose pivot randomly
 - › Need a random number generator
- Choose the first element
 - › Ok if array shuffled, bad if array sorted – worst case for quicksort

Quicksort – in-place partitioning

- › If we use an extra array, partitioning is easy to implement, but not so much easier that it is worth the extra cost of copying the partitioned version back into the original.
- › Partition in-place

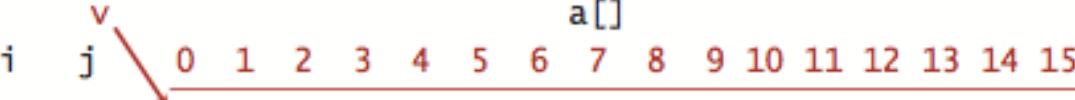
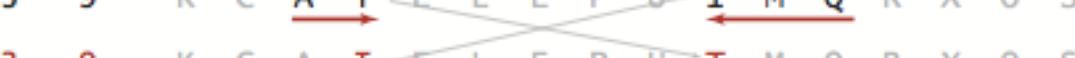
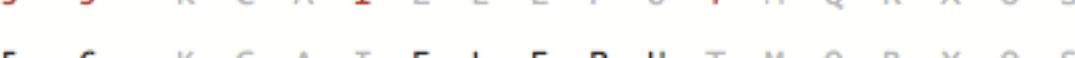
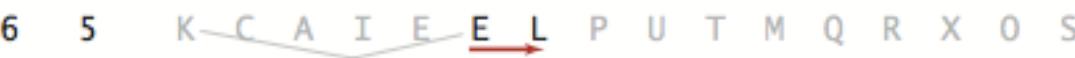
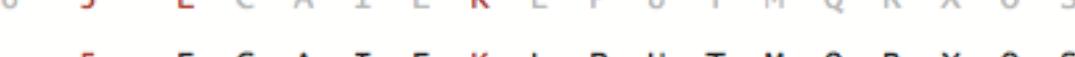
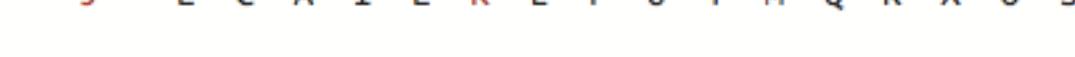
Quicksort – in-place partitioning example

Repeat until i and j pointers cross.

- Scan i from left to right so long as ($a[i] < a[lo]$).
- Scan j from right to left so long as ($a[j] > a[lo]$).
- Exchange $a[i]$ with $a[j]$.



Quicksort – in-place partitioning example

	i	j	v	a[]
initial values	0	16	K R A T E L E P U I M Q C X O S	
scan left, scan right	1	12	R	
exchange	1	12	C	
scan left, scan right	3	9	A	
exchange	3	9	T	
scan left, scan right	5	6	E	
exchange	5	6	L	
scan left, scan right	6	5	E	
final exchange	6	5	K	
result	5			E C A I E K L P U T M Q R X O S

Partitioning trace (array contents before and after each exchange)

Quicksort – in-place partitioning example

Repeat until i and j pointers cross.

- Scan i from left to right so long as ($a[i] < a[lo]$).
- Scan j from right to left so long as ($a[j] > a[lo]$).
- Exchange $a[i]$ with $a[j]$.

When pointers cross.

- Exchange $a[lo]$ with $a[j]$.



Quick sort exercise

A single pass of sorting (no need to continue recursively)

Show pivot value

Show values of i , j , and status of array on each pass

Array:

I L O V E A L G O R I T H M S



Quicksort – partition code

```
private int partition(Comparable[] numbers, int lo, int hi) {  
    int i = lo;  
    int j = hi+1;  
    Comparable pivot = numbers[lo];  
    while(true) {  
        while((numbers[++i].compareTo(pivot) < 0)) {  
            if(i == hi) break;  
        }  
        while((pivot.compareTo(numbers[--j]) < 0)) {  
            if(j == lo) break;  
        }  
        if(i >= j) break;  
        Comparable temp = numbers[i];  
        numbers[i] = numbers[j];  
        numbers[j] = temp;  
    }  
    numbers[lo] = numbers[j];  
    numbers[j] = pivot;  
    return j;  
}
```

Quicksort – example

- › Partitioning one array – need to do this recursively on the array left of j and right of j

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15				
			Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E				
			K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S				
			0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S	
			0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S	
			0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S	
			0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S	
			1			1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
			4			4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
			6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S	
			7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S	
			7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S	
			8			8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
			10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X	
			10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X	
			10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X	
			10			10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
			14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X	
			15			15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
							A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort – recursive code

```
public void sort(Comparable[] numbers) {  
    recursiveQuick(numbers, 0, numbers.length-1);  
}  
  
public void recursiveQuick(Comparable[] numbers, int lo, int hi) {  
    if(hi <= lo) {  
        return;  
    }  
    int pivotPos = partition(numbers, lo, hi);  
    recursiveQuick(numbers, lo, pivotPos-1);  
    recursiveQuick(numbers, pivotPos+1, hi);  
}
```

Quicksort – iterative version?

- › With the help of auxiliary stack

Quicksort – performance

- › How many compares to partition the array of length N ?
- › How many recursive calls? – depth of recursion
- › Best case analysis – for shuffled elements?
- › Worst case analysis – for sorted elements?

- › Polls

Quicksort – best case analysis

What is the number of compares?

			a[]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0	0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O	
2	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O	
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4	4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O	
6	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O	
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8	8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O	
10	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O	
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
14	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort – worst case analysis

What is the number of compares?

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort

- › Make sure to always avoid worst case performance by shuffling the array at the start!
- › Alternatively – pick a random pivot in each subarray
- › Quicksort is therefore a randomized algorithm
 - Uses random numbers to decide what to do next somewhere in its logic

Quicksort – performance

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (n^2)			mergesort ($n \log n$)			quicksort ($n \log n$)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Quicksort - performance

Average case. Expected number of compares is $\sim 1.39 n \lg n$.

- 39% more compares than mergesort.
- Faster than mergesort in practice because of less data movement.

Maths in Sedgwick

Quicksort – properties summary

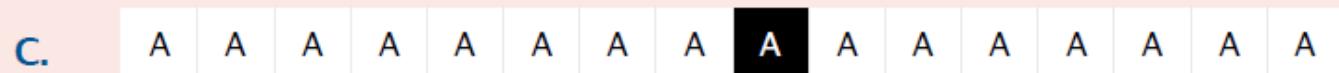
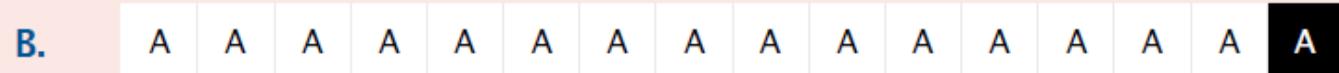
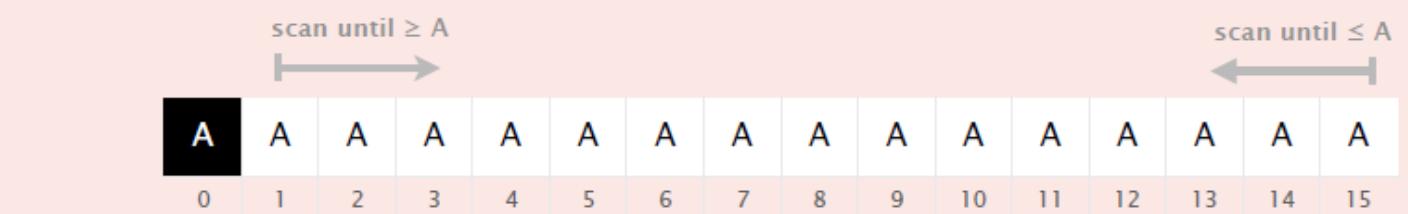
- › Not stable because of long distance swapping.
- › No iterative version (without using a stack).
- › Pure quicksort not good for small arrays.
- › “In-place”, but uses auxiliary storage because of recursive call ($O(\log n)$ space).
- › $O(n \log n)$ average case performance, but $O(n^2)$ worst case performance.

Quicksort improvements

- › Use insertion sort for small arrays
 - Cut off to insertion sort at subarray size ~10
- › Use median for pivot value (median of 3 random items, ie first, last, middle)
- › 3-way quicksort, dual pivot, 3-pivot

Quicksort – all items the same

What is the result of partitioning the following array (stop on equal keys)?



D. *I don't know.*

Quicksort – stop at equal keys

- › qsort() in C bug reported in 1991 – “unbearably slow” for organ-pipe inputs (eg “01233210”)
 - In implementations and textbooks until then
- › N^2 time to sort organ-pipe inputs, and random arrays of 0s and 1s
- › Improvement now: stop scanning if keys are equal



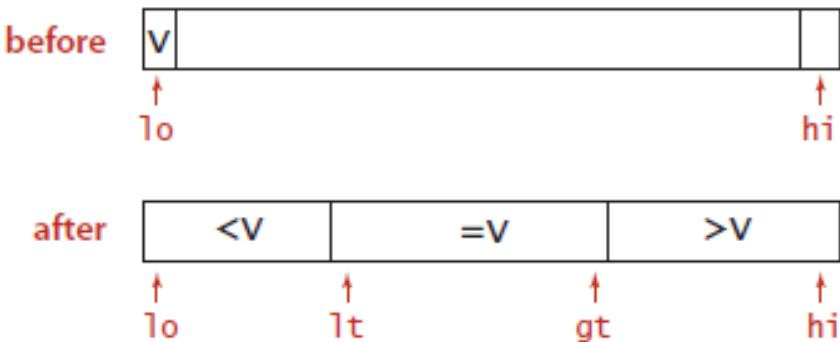
Quicksort – stop at equal keys

- › Problem – if all items equal to pivot are moved to one side of it
 - Consequence $\sim 1/2 n^2$ compares when all keys are equal
- › Stop when keys are equal
 - If all keys are equal, divides the array exactly
 - Why not put all items that are the same as partition item in place? 3-way partitioning

3-way partitioning

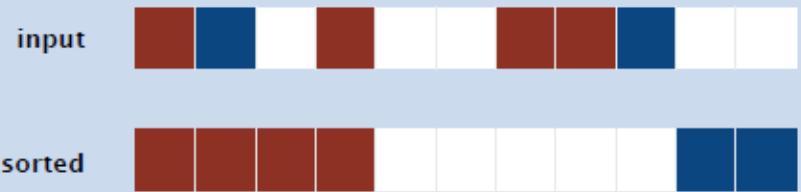
Goal. Partition array into **three** parts so that:

- Entries between lt and gt equal to the partition item.
- No larger entries to left of lt .
- No smaller entries to right of gt .



Dutch national flag problem

Problem. [Edsger Dijkstra] Given an array of n buckets, each containing a red, white, or blue pebble, sort them by color.



Operations allowed.

- $\text{swap}(i, j)$: swap the pebble in bucket i with the pebble in bucket j .
- $\text{color}(i)$: color of pebble in bucket i .

Requirements.

- Exactly n calls to $\text{color}()$.
- At most n calls to $\text{swap}()$.
- Constant extra space.

3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$; increment both lt and i
 - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$; decrement gt
 - $(a[i] == v)$: increment i

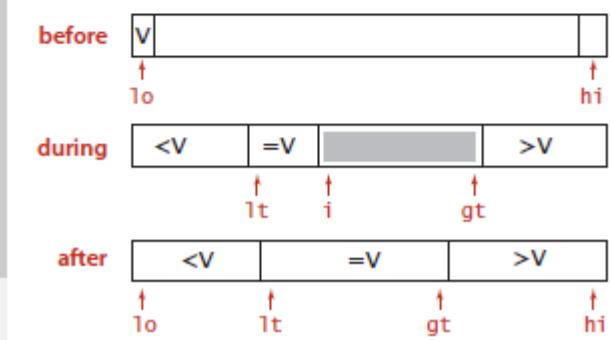


Demo <https://algs4.cs.princeton.edu/lectures/23DemoPartitioning.pdf>

3-way partitioning

- › Improves quick sort when there are duplicate keys

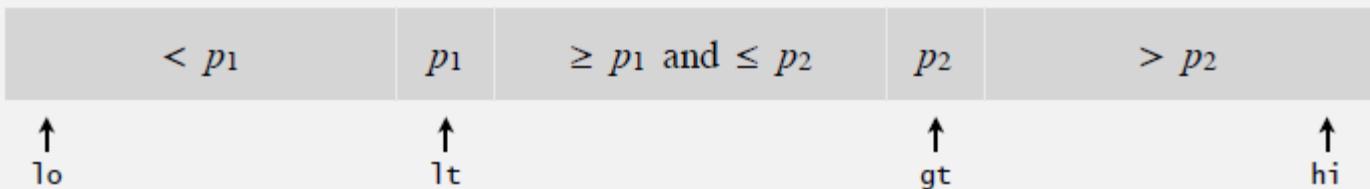
```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    Comparable v = a[lo];
    int i = lo;
    while (i <= gt)
    {
        int cmp = a[i].compareTo(v);
        if      (cmp < 0) exch(a, lt++, i++);
        else if (cmp > 0) exch(a, i, gt--);
        else                  i++;
    }
    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}
```



2-pivot quick sort

Use **two** partitioning items p_1 and p_2 and partition into three subarrays:

- Keys less than p_1 .
- Keys between p_1 and p_2 .
- Keys greater than p_2 .



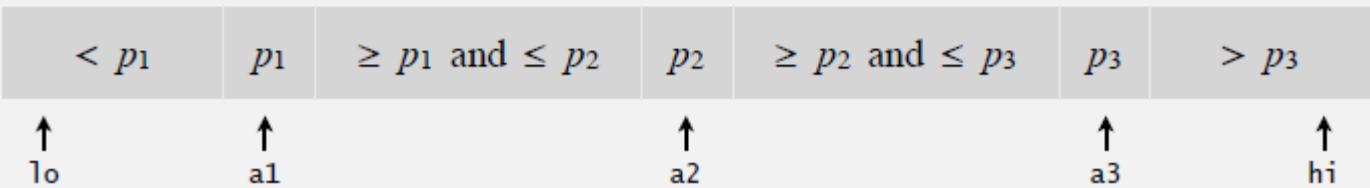
Recursively sort three subarrays.

3-pivot quick sort

Three-pivot quicksort

Use **three** partitioning items p_1 , p_2 , and p_3 and partition into four subarrays:

- Keys less than p_1 .
- Keys between p_1 and p_2 .
- Keys between p_2 and p_3 .
- Keys greater than p_3 .



Demos

- › <https://algs4.cs.princeton.edu/lectures/23DemoPartitioning.pdf>
- › Quicksort
- › 3-way partitioning
- › Dual pivot partitioning

Quicksort – cache improvements

- › Principle of locality
 - the same values, or related storage locations, are frequently accessed
 - Temporal locality
 - › If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future
 - Spatial locality
 - › If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future -> pre-fetch arrays
 - Predictability of memory access
 - Implications for caching
 - › cache – stores data “nearer” to processor so that it can be accessed quicker in the future
- › 2-pivot and 3-pivot have smaller number of cache misses and smaller number of recursive calls to a subproblem larger than the size of a cache block
- › Multi-Pivot Quicksort: Theory and Experiments - by Kushagra, López-Ortiz, Munro, and Qiao
 - Original paper - <http://pubs.siam.org/doi/pdf/10.1137/1.9781611973198.6>
 - Discussion: <https://cs.stanford.edu/~rishig/courses/ref/l11a.pdf>

Caching improvements

Why do 2-pivot (and 3-pivot) quicksort perform better than 1-pivot?

- A. Fewer compares.
- B. Fewer exchanges. # entries scanned is a good proxy for cache performance when comparing quicksort variants
- C. Fewer cache misses.

partitioning	compares	exchanges	entries scanned
1-pivot	$2 n \ln n$	$0.333 n \ln n$	$2 n \ln n$
median-of-3	$1.714 n \ln n$	$0.343 n \ln n$	$1.714 n \ln n$
2-pivot	$1.9 n \ln n$	$0.6 n \ln n$	$1.6 n \ln n$
3-pivot	$1.846 n \ln n$	$0.616 n \ln n$	$1.385 n \ln n$

Reference: Analysis of Pivot Sampling in Dual-Pivot Quicksort by Wild–Nebel–Martínez

Bottom line. Caching can have a significant impact on performance.

Merge vs quick

- › In Java, `Arrays.sort()` uses **QuickSort** for sorting primitives and **MergeSort** for sorting Arrays of Objects. This is because, merge sort is stable, so it won't reorder elements that are equal.
 - Why does it matter for Objects and not for primitive data types?
- › QuickSort in java
 - 2-pivot since 2009
- › MergeSort in java
 - Timsort

Sort algorithms summary

- › Use system sort - `Arrays.sort();` - usually good enough
- › What to consider when picking an algorithm?

Compare performance to system sort in your assignment?

Comparator interface

- › Comparable interface
 - Uses natural order to compare things
 - Can override method `compareTo()` if want custom-defined criteria
- › But what if we have Objects we want to compare according to multiple custom-defined criteria?
- › Comparator interface
 - Can create multiple classes implementing Comparator and override compare method
 - Custom ordering
 - To use with system sort, pass as a second argument to `Array.sort(a, new MyCustomOrder());`

Sorting algorithms summary

	inplace?	stable?	best	average	worst	remarks
selection	✓		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges
insertion	✓	✓	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small n or partially ordered
shell	✓		$n \log_3 n$?	$c n^{3/2}$	tight code; subquadratic
merge		✓	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
timsort		✓	n	$n \lg n$	$n \lg n$	improves mergesort when preexisting order
quick	✓		$n \lg n$	$2 n \ln n$	$\frac{1}{2} n^2$	$n \log n$ probabilistic guarantee; fastest in practice
3-way quick	✓		n	$2 n \ln n$	$\frac{1}{2} n^2$	improves quicksort when duplicate keys
heap	✓		$3 n$	$2 n \lg n$	$2 n \lg n$	$n \log n$ guarantee; in-place
?	✓	✓	n	$n \lg n$	$n \lg n$	holy sorting grail

Quick algorithms exercise

- › Which algorithm would work best to sort data as it arrives, one piece at a time, perhaps from a network?
 1. Mergesort
 2. Selection sort
 3. Quicksort
 4. Insertion sort

Another quick question

- › Which algorithm would you use to sort 1 million of 32-bit integers?
 1. Mergesort
 2. Selection sort
 3. Quicksort
 4. Insertion sort
 5. None of the above

https://www.youtube.com/watch?v=k4RRi_ntQc8

Quick sort exercise

Illustrate quick sort by showing at each pass through the array the swaps that happen and circle the final location of the pivot element

Pivot = array [0]

i ++ until find element > than pivot

j - - until find element < than

piv

5	4	7	2	2	8	1	9
---	---	---	---	---	---	---	---

Quick sort exercise

Illustrate quick sort by showing at each pass through the array the swaps that happen and circle the final location of the pivot element

Pivot = array [0]

i ++ until find element > than pivot

j - - until find element < than

piv

2	2	2	2	2	2	2	2
---	---	---	---	---	---	---	---

Quick sort exercise 2

Illustrate quick sort by showing at each pass through the array the swaps that happen and circle the final location of the pivot element

Pivot = array [0]

i ++ until find element \geq than pivot

j - - until find element \leq than

piv	2	2	2	2	2	2	2	2
-----	---	---	---	---	---	---	---	---



CSU22012: Data Structures and Algorithms II

Graphs

Ivana.Dusparic@scss.tcd.ie

Outline

- › Undirected graphs
 - Depth-first search (DFS) – path finding
 - Breadth-first search (BFS) – shortest path
- › Directed graphs – DFS, BFS
- › Directed acyclic graphs
 - Topological sort
- › Shortest path finding
 - Dijkstra's algorithm

Outline

- › Minimum spanning trees
 - Prim's and Kruskal's algorithms – greedy algorithms
- › Shortest paths
 - Single source shortest path
 - › Topological sort – acyclic graphs
 - › Dijkstra – non negative weights
 - › Bellman-Ford – non-negative cycles
 - Single-pair shortest path
 - › A* search algorithm
 - All pairs shortest path
 - › Floyd-Warshall

Outline

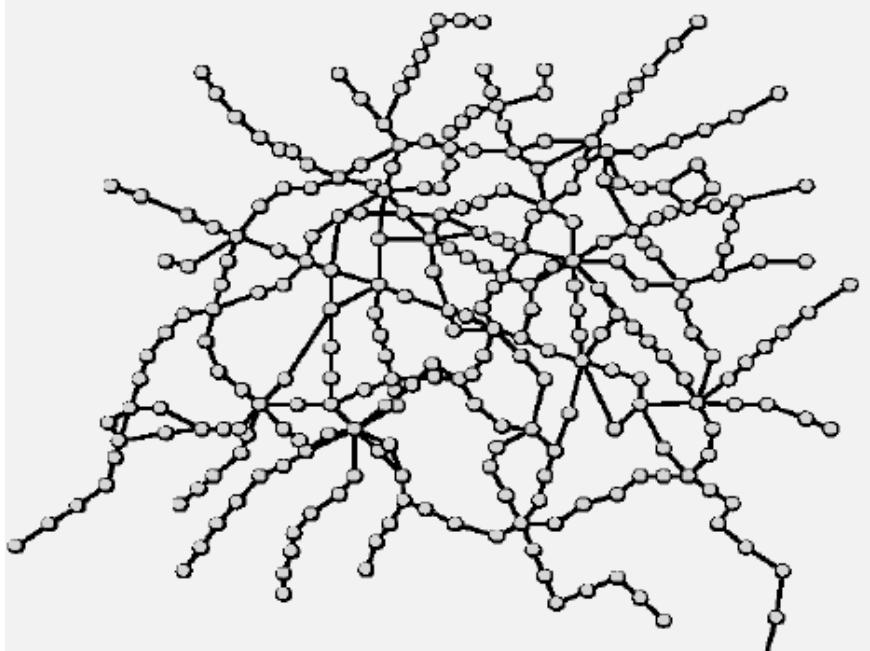
- › Dynamic programming examples
 - Bellman-Ford and Floyd-Warshall
- › Network flow
 - Positive edge (capacity) directed graph with a source and sink
 - Maxflow problem – flow of maximum capacity
 - Ford-Fulkerson algorithm

Background and applications

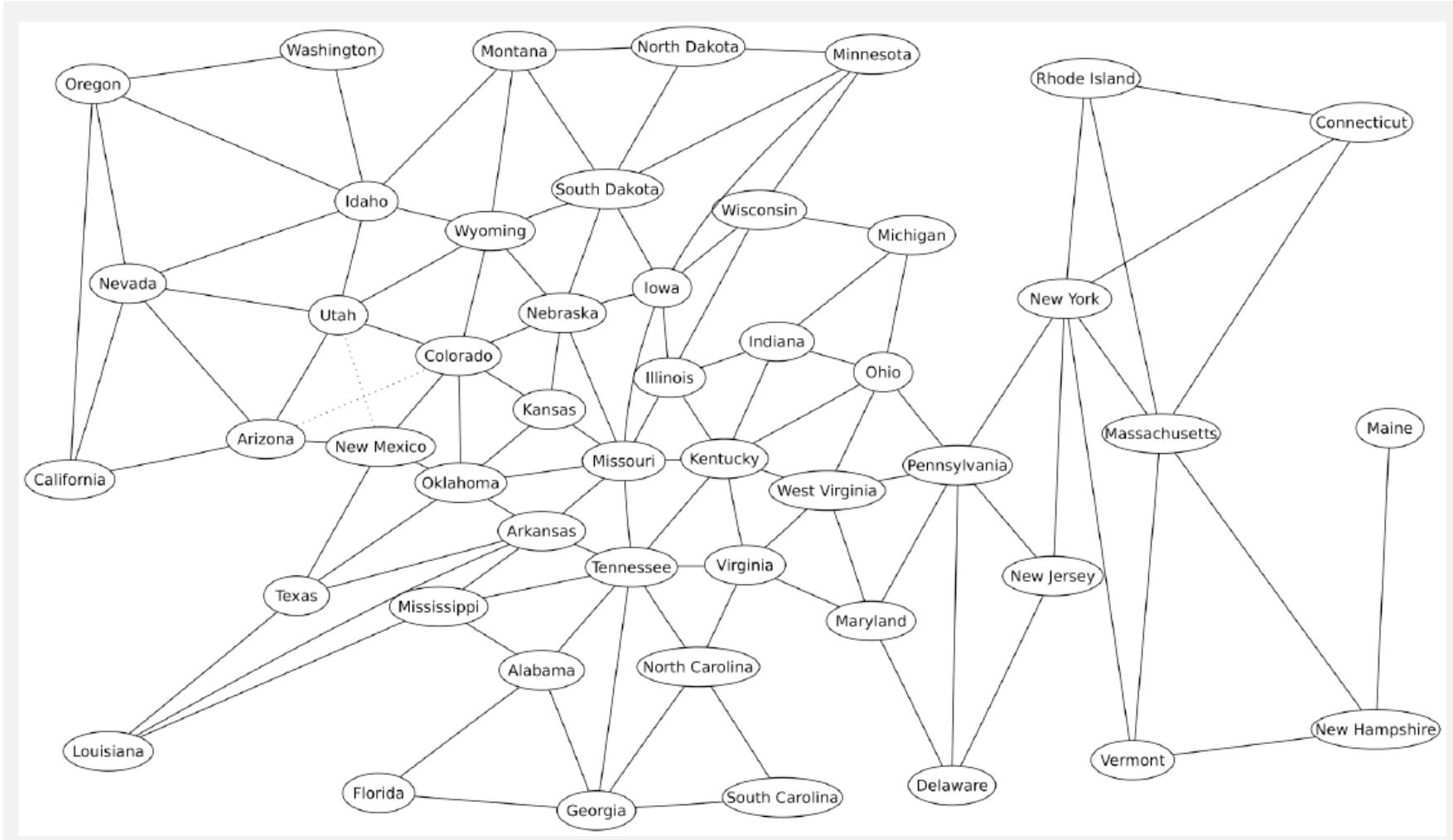
- › Graph = a set of vertices connected pairwise by edges
- › Thousands of practical applications.
- › Hundreds of graph algorithms known.
- › Interesting and broadly useful abstraction.
- › Challenging branch of computer science and discrete math

graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	intersection	street
internet	class C network	connection
game	board position	legal move
social relationship	person	friendship
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

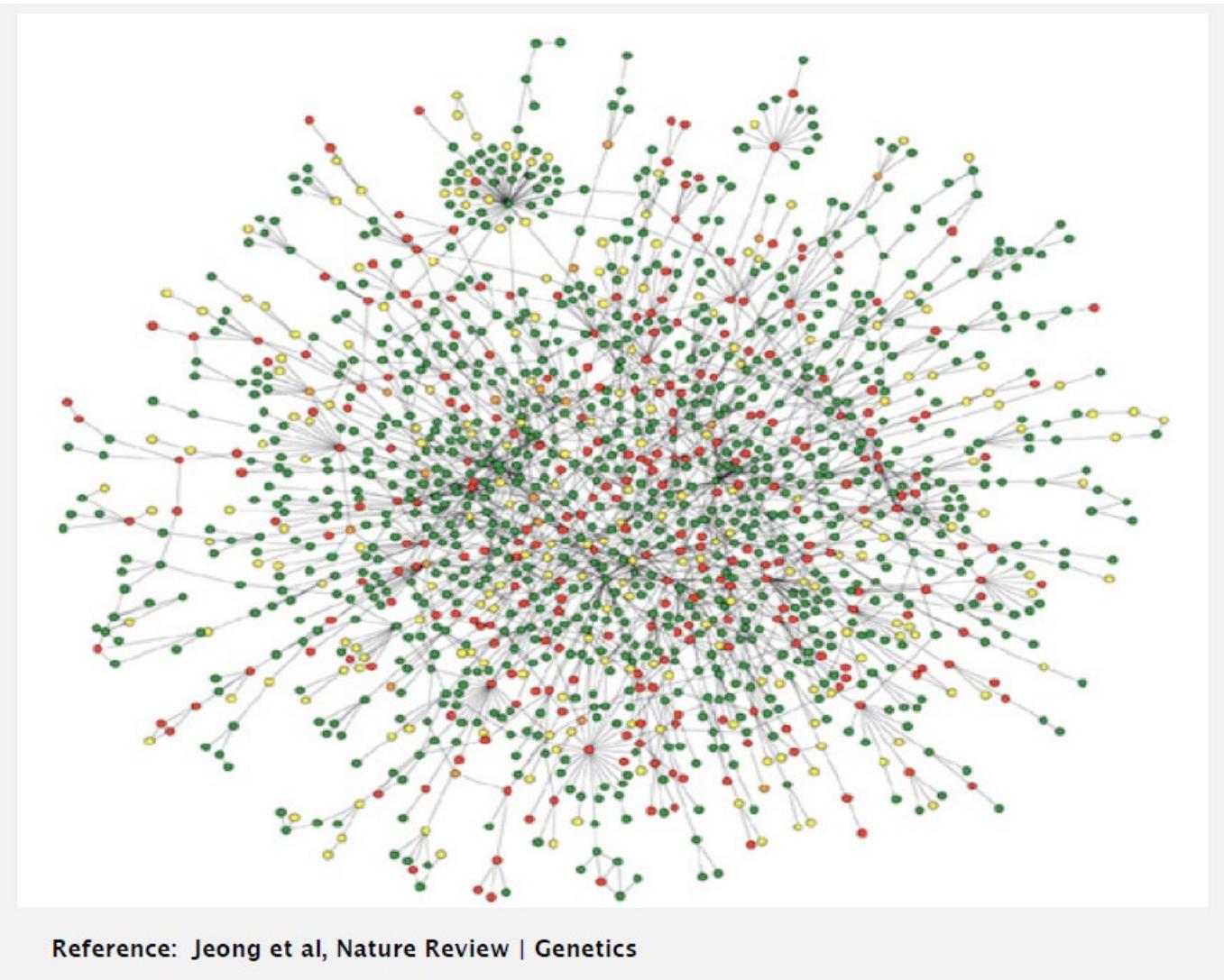
Transport networks



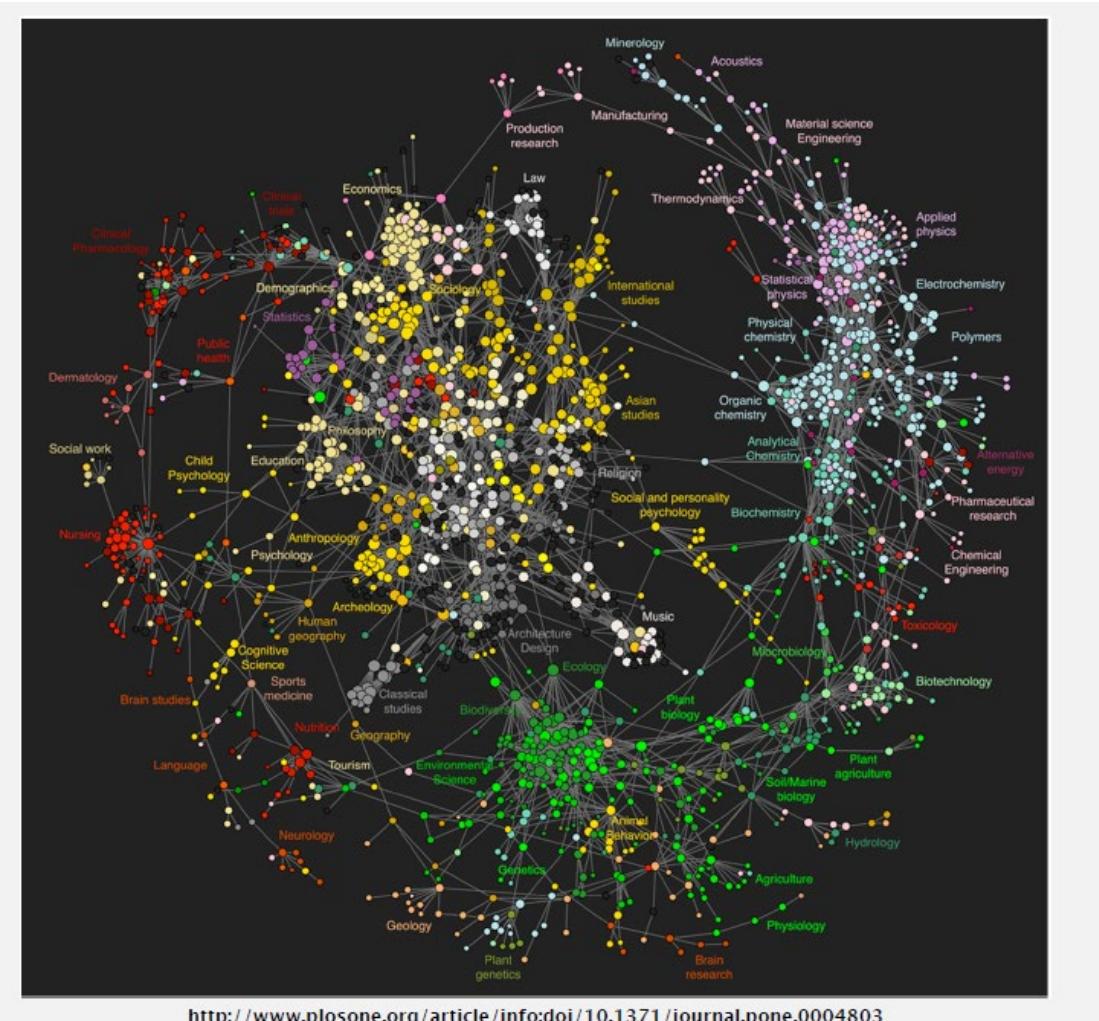
Borders graph – American states



Protein interaction network



Science clickstreams – citation networks



Facebook connections



"Visualizing Friendships" by Paul Butler

Health studies

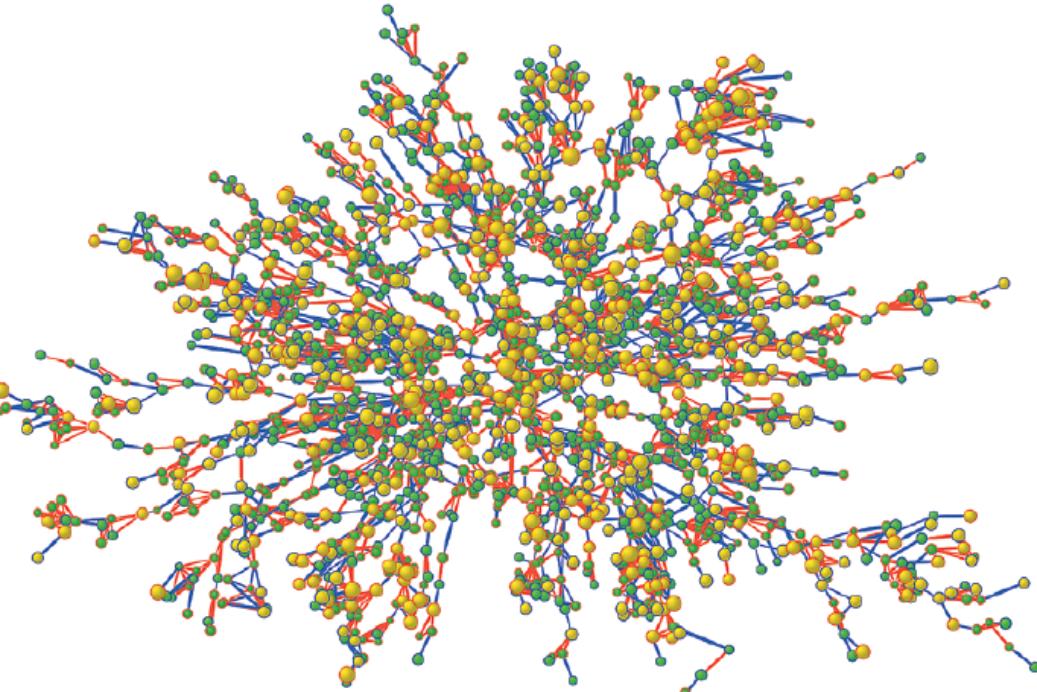
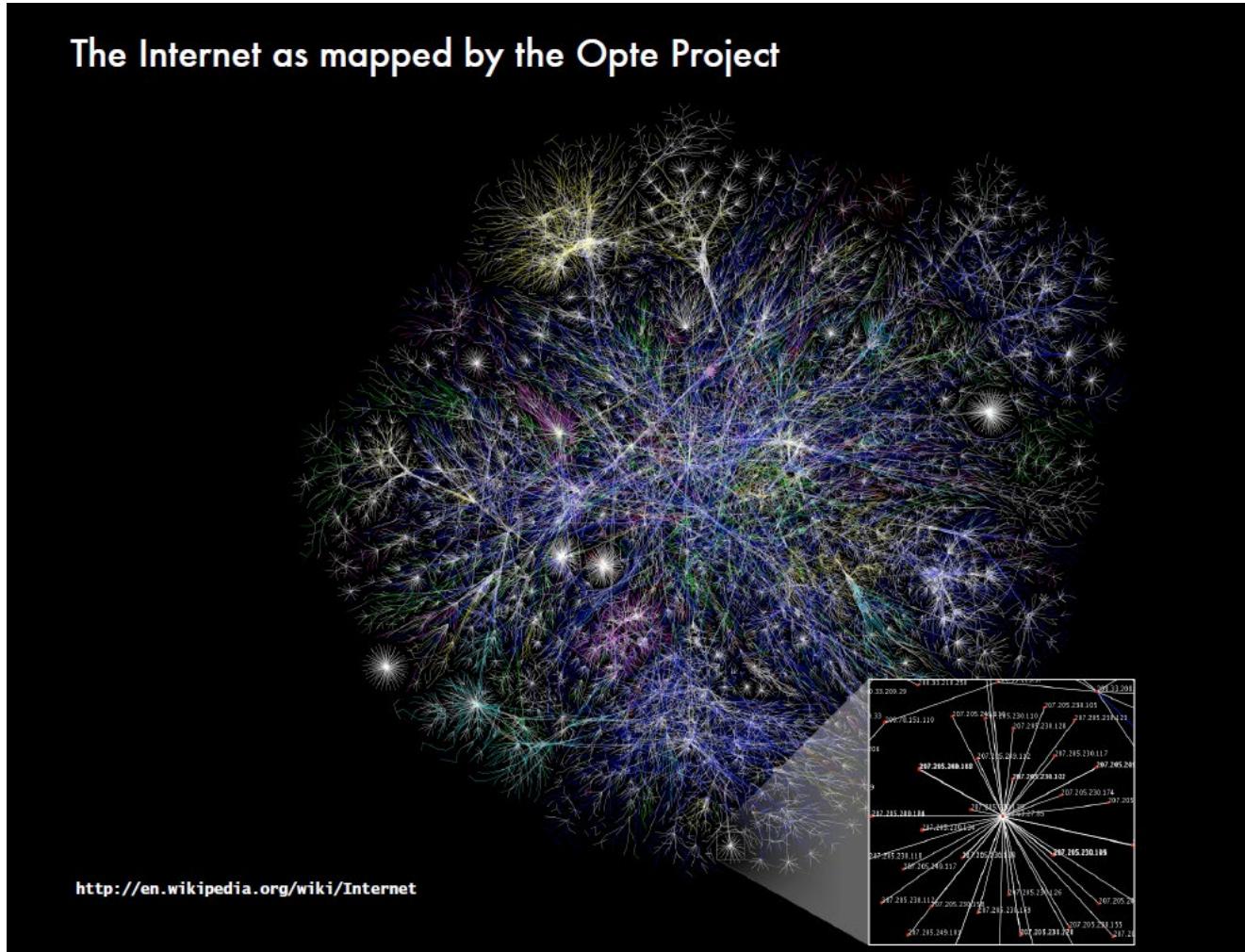


Figure 1. Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000.
Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index, ≥ 30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.

Internet links

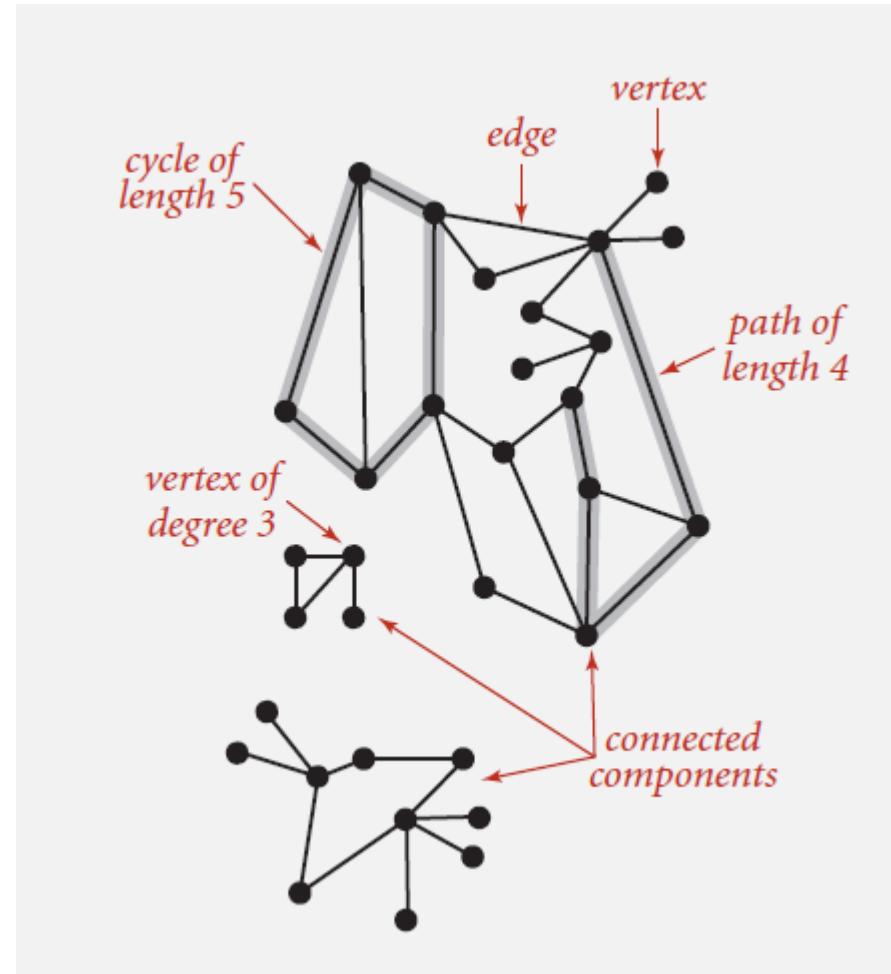


How connected are we?

- › Small-world experiment by Milgram
- › 6 degrees of separation
 - 3.57 on facebook
- › 6 degrees of Kevin Bacon
- › Erdos number

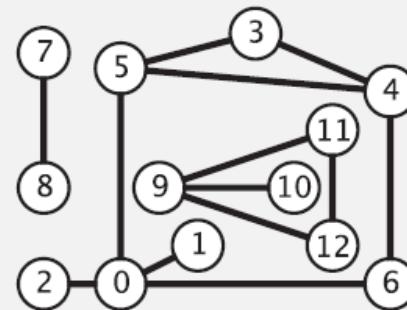
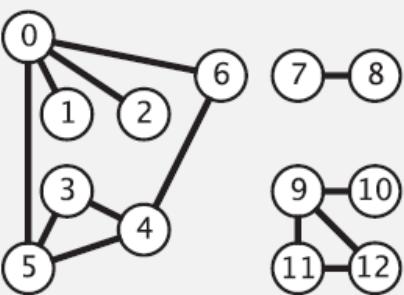
Graph terminology

- › Path - Sequence of vertices connected by edges.
- › Cycle - Path whose first and last vertices are the same.
- › Two vertices are connected if there is a path between them



Graph representation

Graph drawing. Provides intuition about the structure of the graph.



two drawings of the same graph

Caveat. Intuition can be misleading.

Graph API

```
public class Graph
```

```
    Graph(int V)
```

create an empty graph with V vertices

```
    Graph(In in)
```

create a graph from input stream

```
    void addEdge(int v, int w)
```

add an edge $v-w$

```
    Iterable<Integer> adj(int v)
```

vertices adjacent to v

```
    int V()
```

number of vertices

```
    int E()
```

number of edges

```
In in = new In(args[0]);  
Graph G = new Graph(in);
```

read graph from
input stream

```
for (int v = 0; v < G.V(); v++)  
    for (int w : G.adj(v))  
        StdOut.println(v + "-" + w);
```

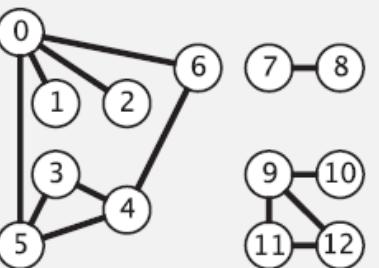
print out each
edge (twice)

Graph API: sample client

Graph input format.

tinyG.txt
V → 13
13 ← E

0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3



```
% java Test tinyG.txt  
0-6  
0-2  
0-1  
0-5  
1-0  
2-0  
3-5  
3-4  
:  
12-11  
12-9
```

```
In in = new In(args[0]);  
Graph G = new Graph(in);  
  
for (int v = 0; v < G.V(); v++)  
    for (int w : G.adj(v))  
        StdOut.println(v + "-" + w);
```

read graph from
input stream

print out each
edge (twice)

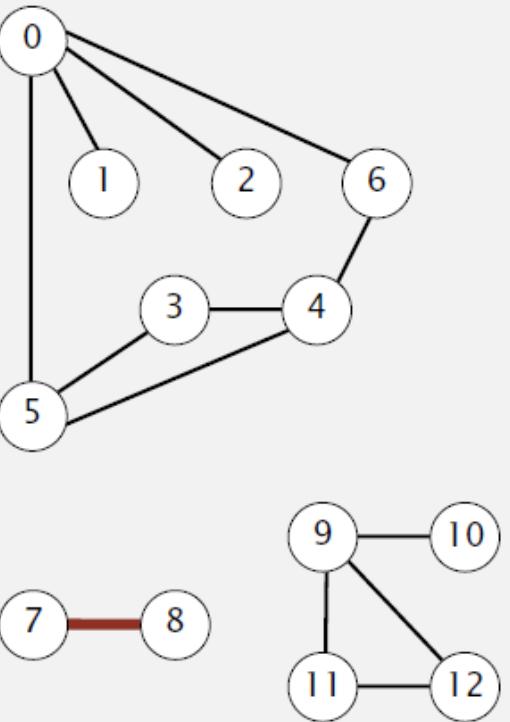
Typical graph-processing code

```
public class Graph
{
    Graph(int V)                      create an empty graph with V vertices
    Graph(In in)                      create a graph from input stream
    void addEdge(int v, int w)          add an edge v-w
    Iterable<Integer> adj(int v)       vertices adjacent to v
    int V()                           number of vertices
    int E()                           number of edges
}
```

```
// degree of vertex v in graph G
public static int degree(Graph G, int v)
{
    int degree = 0;
    for (int w : G.adj(v))
        degree++;
    return degree;
}
```

Set-of-edges graph representation

Maintain a list of the edges (linked list or array).

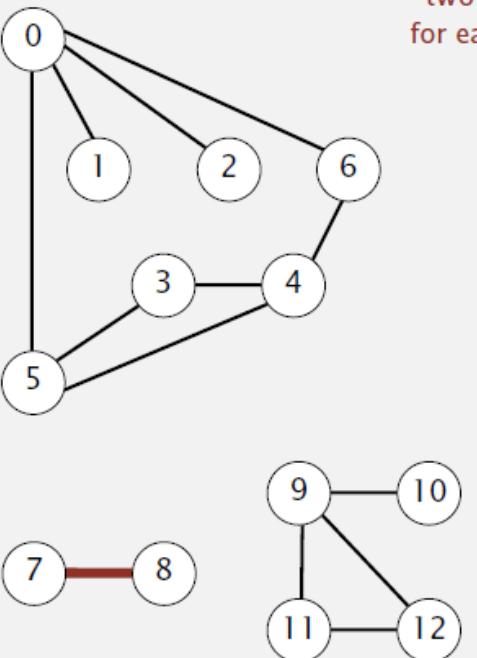


0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

Q. How long to iterate over vertices adjacent to v ?

Adjacency-matrix graph representation

Maintain a two-dimensional V -by- V boolean array;
for each edge $v-w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$.



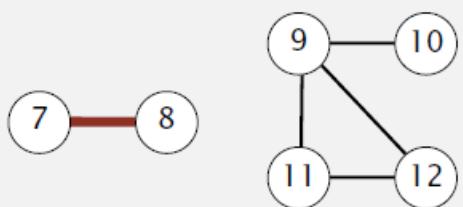
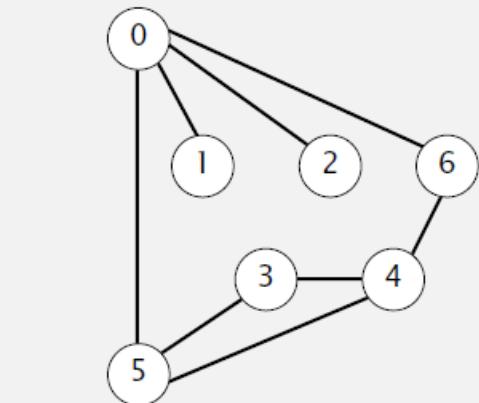
two entries
for each edge

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	1	0	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

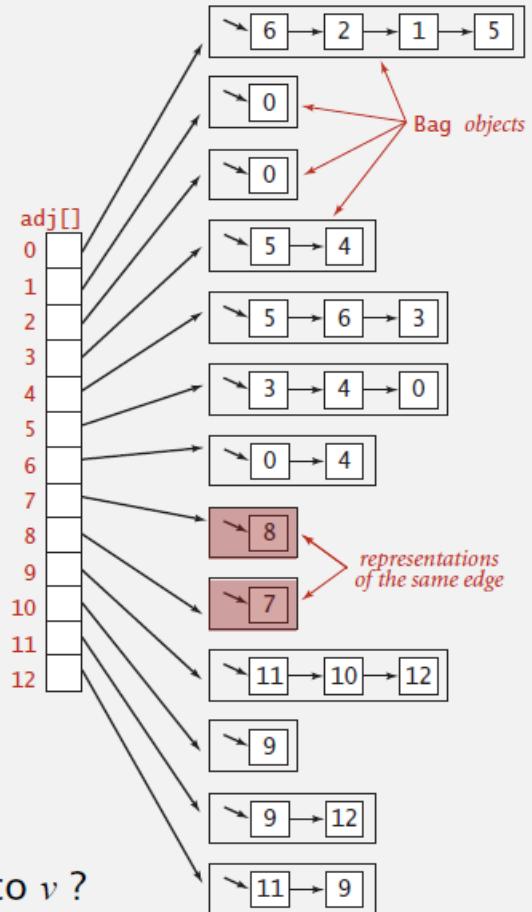
Q. How long to iterate over vertices adjacent to v ?

Adjacency-list graph representation

Maintain vertex-indexed array of lists.



Q. How long to iterate over vertices adjacent to v ?

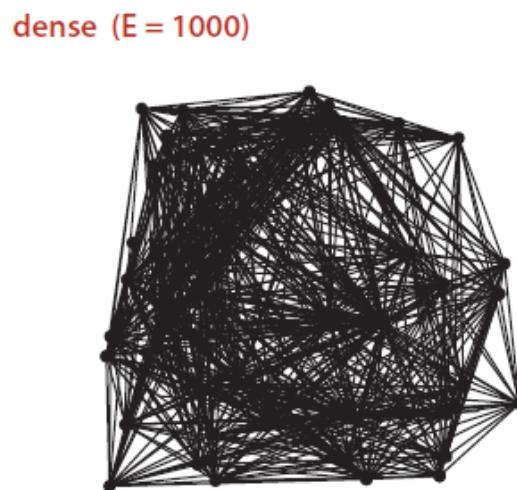
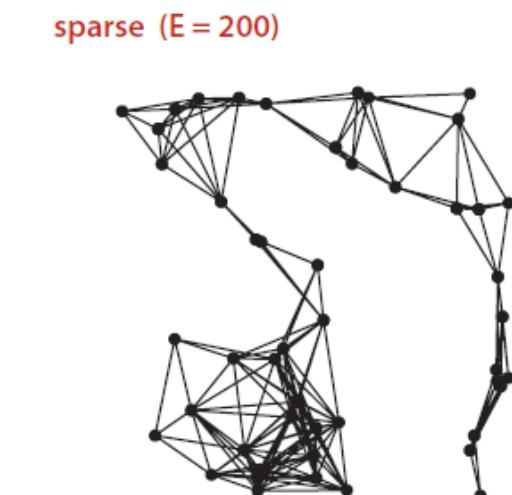


Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v .
- Real-world graphs tend to be **sparse**.

huge number of vertices,
small average vertex degree



Two graphs ($V = 50$)

Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v .
- Real-world graphs tend to be **sparse**.

huge number of vertices,
small average vertex degree

representation	space	add edge	edge between v and w ?	iterate over vertices adjacent to v ?
list of edges	E	1	E	E
adjacency matrix	V^2	1^*	1	V
adjacency lists	$E + V$	1	$degree(v)$	$degree(v)$

* disallows parallel edges

Adjacency-list graph representation: Java implementation

```
public class Graph
{
    private final int V;
    private Bag<Integer>[] adj; ← adjacency lists  
          ( using Bag data type )
```

```
    public Graph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V]; ← create empty graph
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }
```

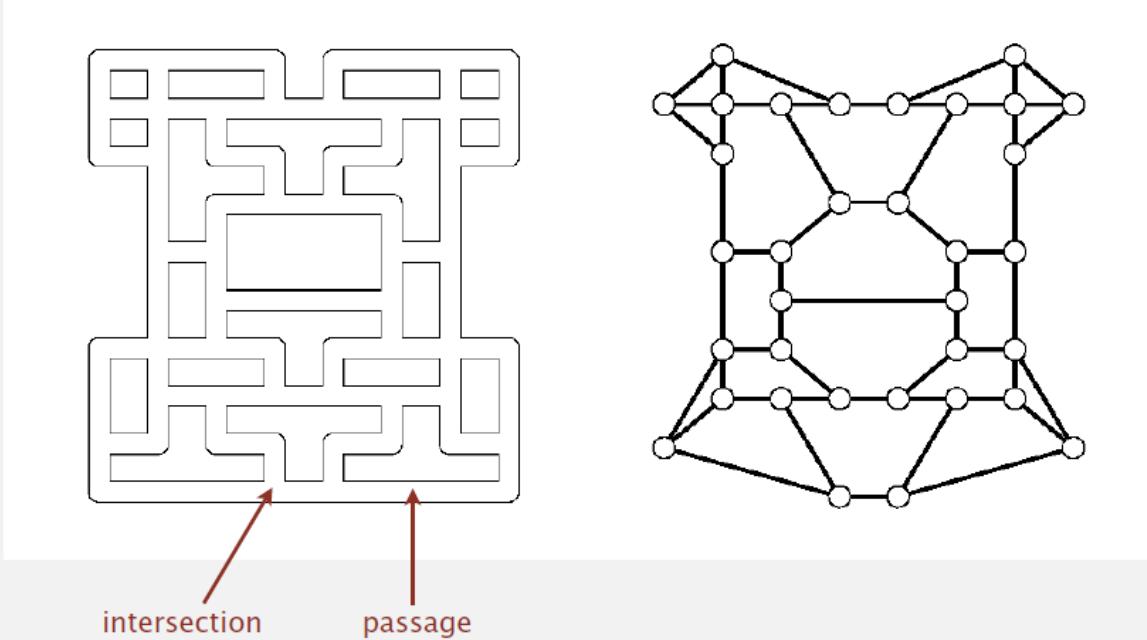
```
    public void addEdge(int v, int w)
    {
        adj[v].add(w); ← add edge v-w
        adj[w].add(v); ← (parallel edges and
                         self-loops allowed)
    }
```

```
    public Iterable<Integer> adj(int v) ← iterator for vertices adjacent to v
    { return adj[v]; }
}
```

Depth First Search (DFS)

Maze graph.

- Vertex = intersection.
- Edge = passage.

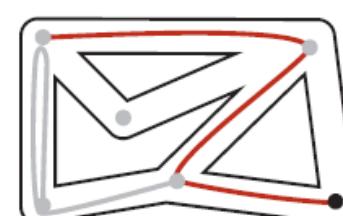
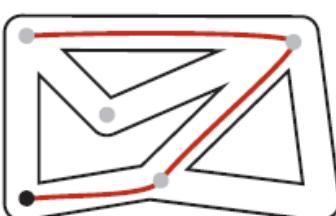
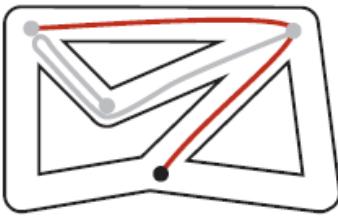
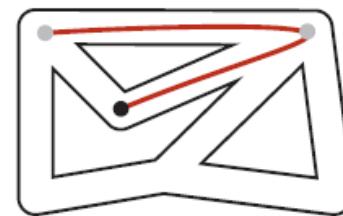
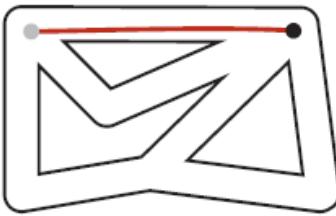
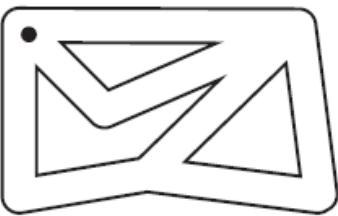


Goal. Explore every intersection in the maze.

Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.



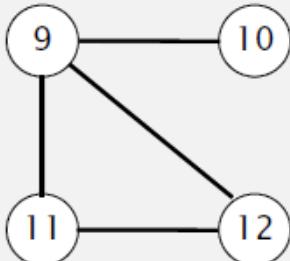
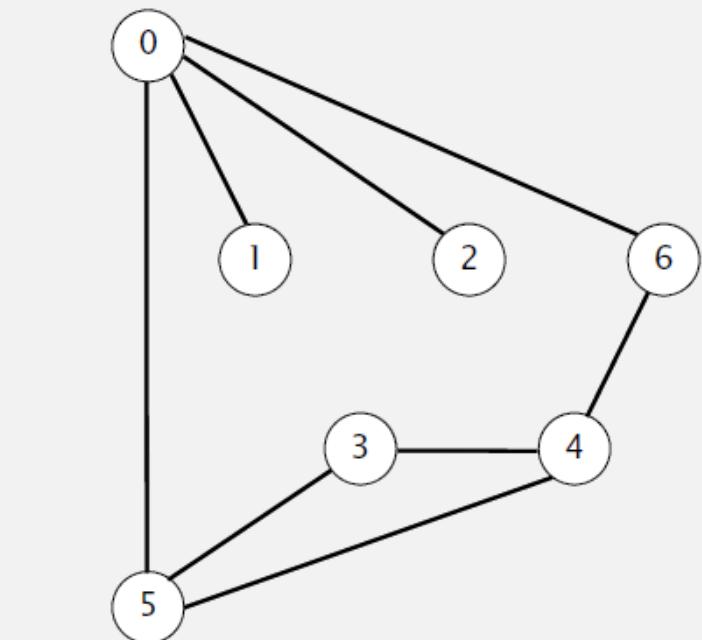
Depth-first search (DFS)

- › Undirected or directed graphs
 - Undirected for now
 - › Find all vertices connected to a given source vertex
 - › Find a path between 2 vertices
-
- › Mark vertex v as visited
 - › Recursively visit all unmarked vertices adjacent to v (or pointing from v, in directed graphs)
-
- › boolean marked [] – list visited vertices
 - › Integer edgeTo[] – $\text{edgeTo}[w]=v$, means edge v-w taken to visit W (for the first time)

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



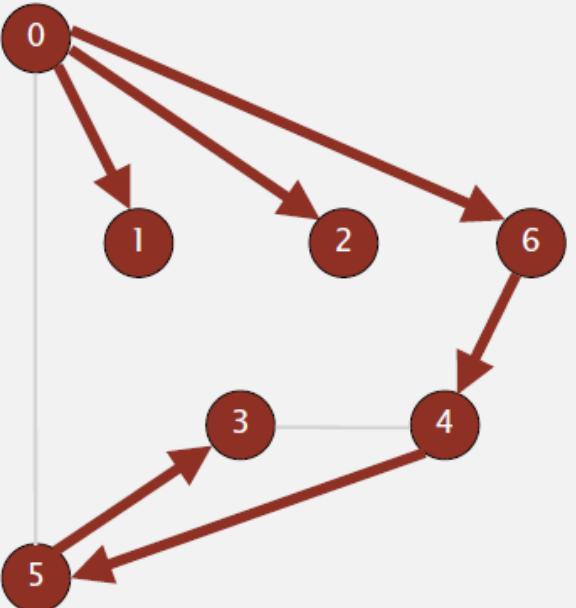
tinyG.txt
V → 13
13 ← E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3

graph G

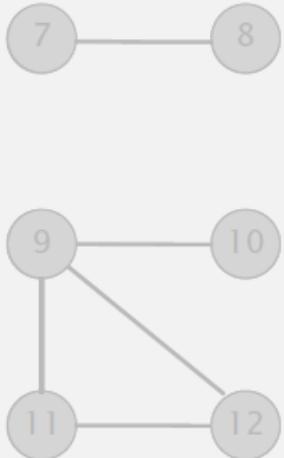
Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



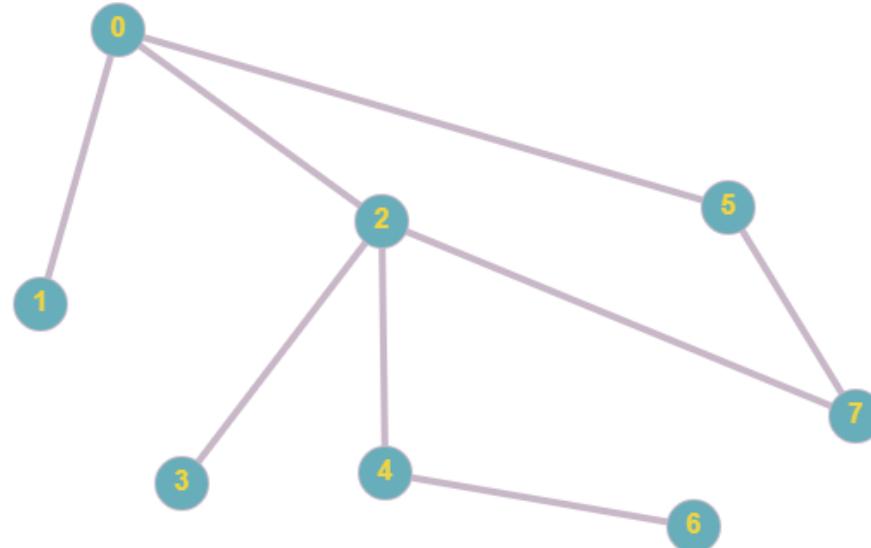
vertices reachable from 0



<code>v</code>	<code>marked[]</code>	<code>edgeTo[]</code>
0	T	-
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

DFS exercise

v	marked[]	edgeTo[]
0		
1		
2		
3		
4		
5		
6		
7		



Source vertex: 0

Provide the trace in which vertices were visited

<http://graphonline.ru/en/> - draw your own graphs and run algorithms

Design pattern for graph processing

Design pattern. Decouple graph data type from graph processing.

- Create a Graph object.
- Pass the Graph to a graph-processing routine.
- Query the graph-processing routine for information.

```
public class Paths
```

Paths(Graph G, int s)	<i>find paths in G from source s</i>
-----------------------	--------------------------------------

boolean hasPathTo(int v)	<i>is there a path from s to v?</i>
--------------------------	-------------------------------------

Iterable<Integer> pathTo(int v)	<i>path from s to v; null if no such path</i>
---------------------------------	---

```
Paths paths = new Paths(G, s);
for (int v = 0; v < G.V(); v++)
    if (paths.hasPathTo(v))
        StdOut.println(v);
```

print all vertices
connected to s

Depth-first search: data structures

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

Data structures.

- Boolean array `marked[]` to mark visited vertices.
- Integer array `edgeTo[]` to keep track of paths.
 $(\text{edgeTo}[w] == v)$ means that edge $v-w$ taken to visit w for first time
- Function-call stack for recursion.

Depth-first search: Java implementation

```
public class DepthFirstPaths
{
    private boolean[] marked;
    private int[] edgeTo;
    private int s;

    public DepthFirstPaths(Graph G, int s)
    {
        ...
        dfs(G, s);
    }

    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w])
            {
                dfs(G, w);
                edgeTo[w] = v;
            }
    }
}
```

marked[v] = true
if v connected to s

edgeTo[v] = previous vertex on path from s to v

initialize data structures

find vertices connected to s

recursive DFS does the work

Depth-first search: properties

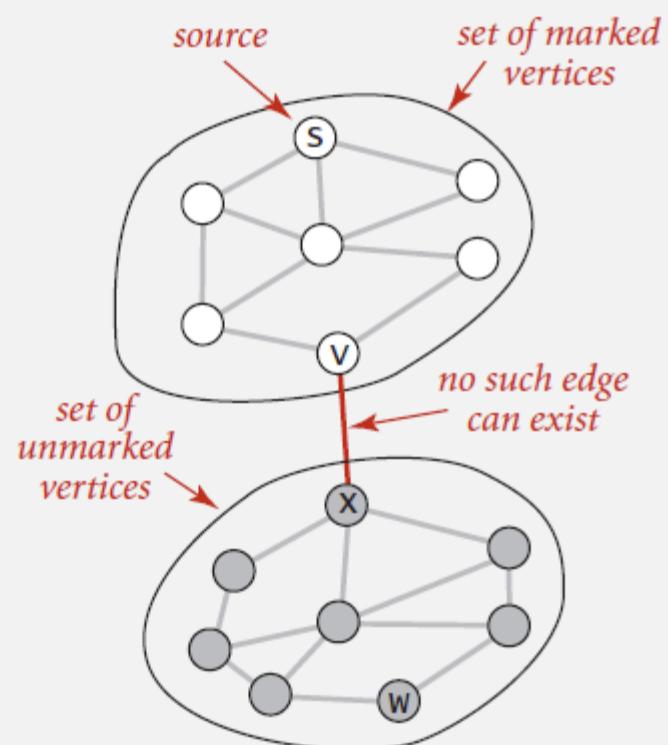
Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees (plus time to initialize the `marked[]` array).

Pf. [correctness]

- If w marked, then w connected to s (why?)
- If w connected to s , then w marked.
(if w unmarked, then consider last edge on a path from s to w that goes from a marked vertex to an unmarked one).

Pf. [running time]

Each vertex connected to s is visited once.



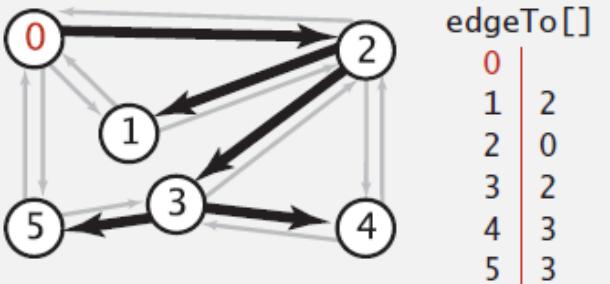
Depth-first search: properties

Proposition. After DFS, can check if vertex v is connected to s in constant time and can find $v-s$ path (if one exists) in time proportional to its length.

Pf. `edgeTo[]` is parent-link representation of a tree rooted at vertex s .

```
public boolean hasPathTo(int v)
{  return marked[v];  }

public Iterable<Integer> pathTo(int v)
{
    if (!hasPathTo(v)) return null;
    Stack<Integer> path = new Stack<Integer>();
    for (int x = v; x != s; x = edgeTo[x])
        path.push(x);
    path.push(s);
    return path;
}
```



Depth-first search application: flood fill

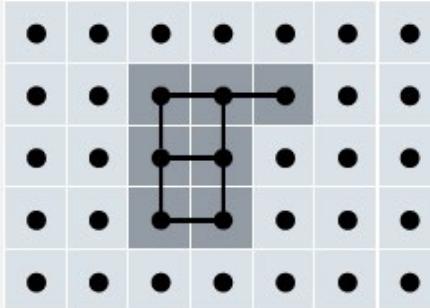
Challenge. Flood fill (Photoshop magic wand).

Assumptions. Picture has millions to billions of pixels.

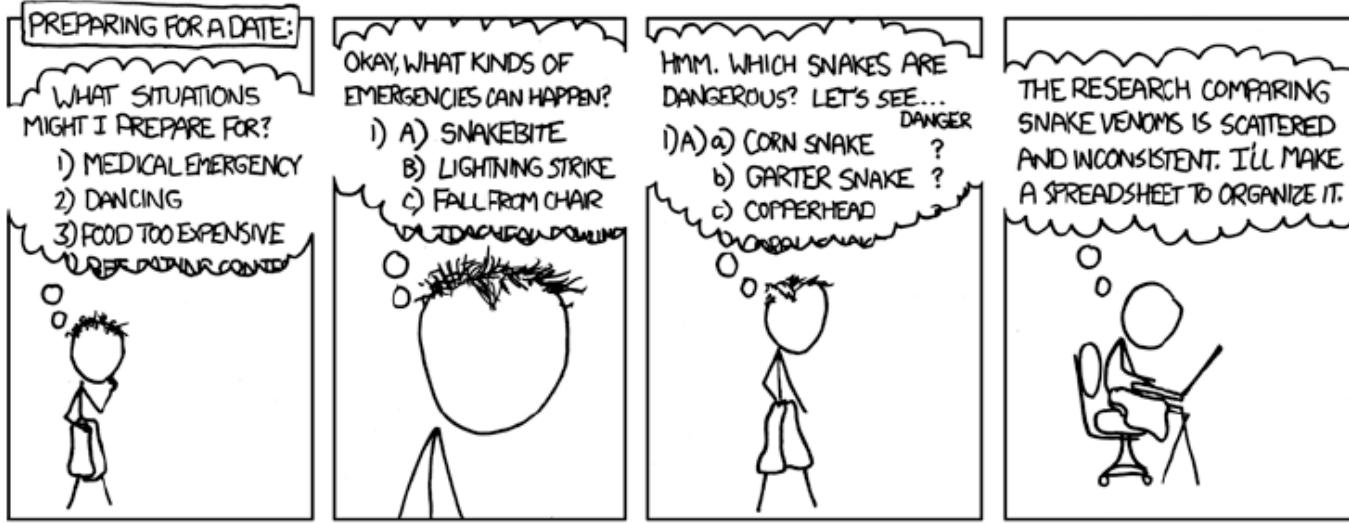


Solution. Build a grid graph (implicitly).

- Vertex: pixel.
- Edge: between two adjacent gray pixels.
- Blob: all pixels connected to given pixel.



Depth-first search application: preparing for a date



I REALLY NEED TO STOP
USING DEPTH-FIRST SEARCHES.



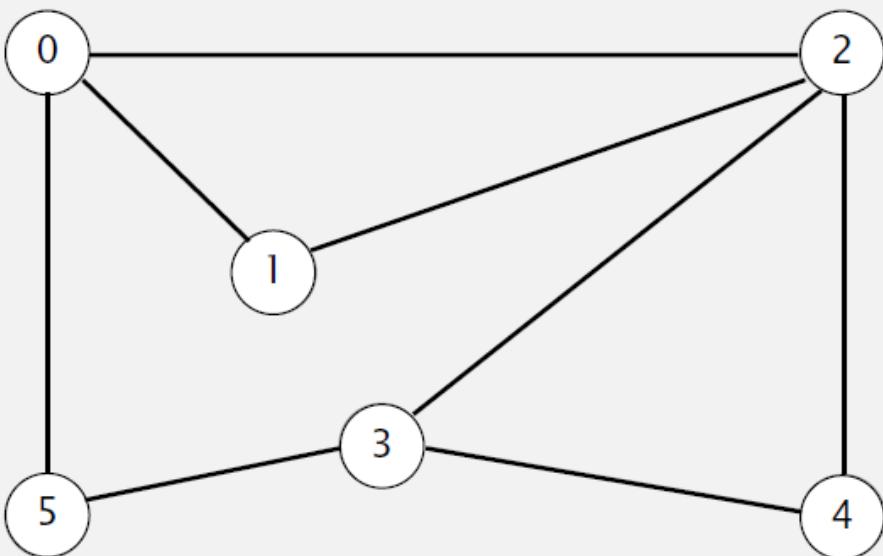
<http://xkcd.com/761/>

Breadth First Search (BFS)

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



tinyCG.txt

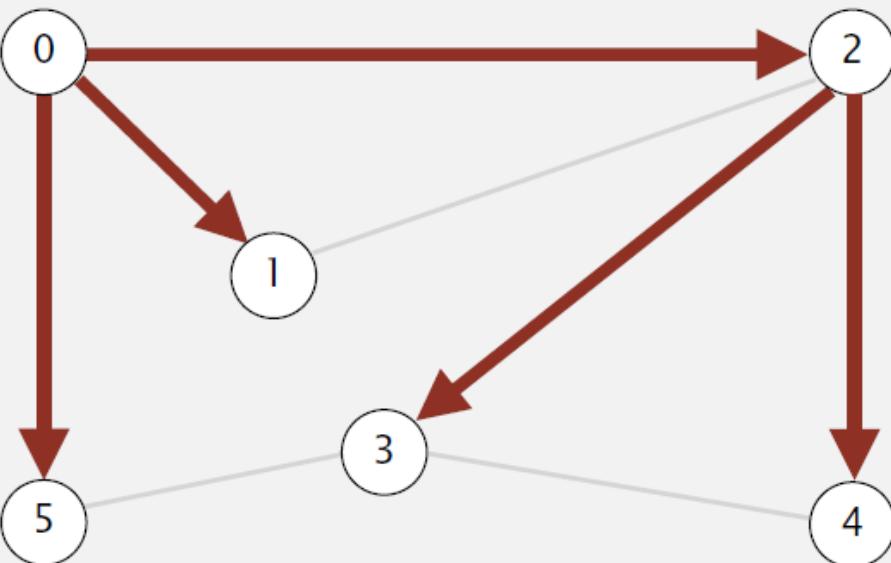
$V \rightarrow$ 6
8 $\leftarrow E$

0	5
2	4
2	3
1	2
0	1
3	4
3	5
0	2

Breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



v	edgeTo[]	distTo[]
0	-	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

Breadth-first search

Repeat until queue is empty:

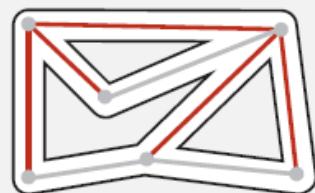
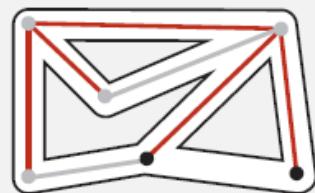
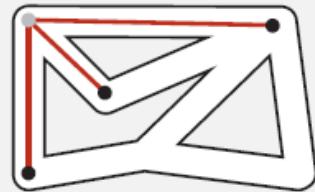
- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

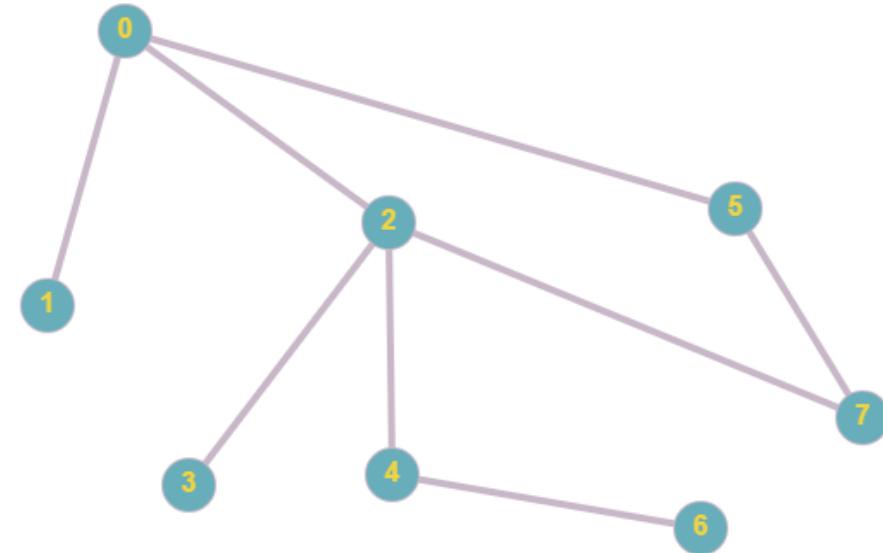
Repeat until the queue is empty:

- remove the least recently added vertex v
 - add each of v 's unvisited neighbors to the queue,
and mark them as visited.
-



BFS Exercise

v	marked[]	edgeTo[]	distanceTo[]
0			
1			
2			
3			
4			
5			
6			
7			



Source vertex: 0

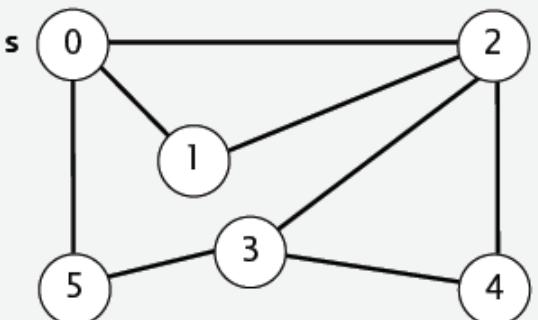
Fill in the table and provide trace of the queue content – order in which vertices were visited

Breadth-first search properties

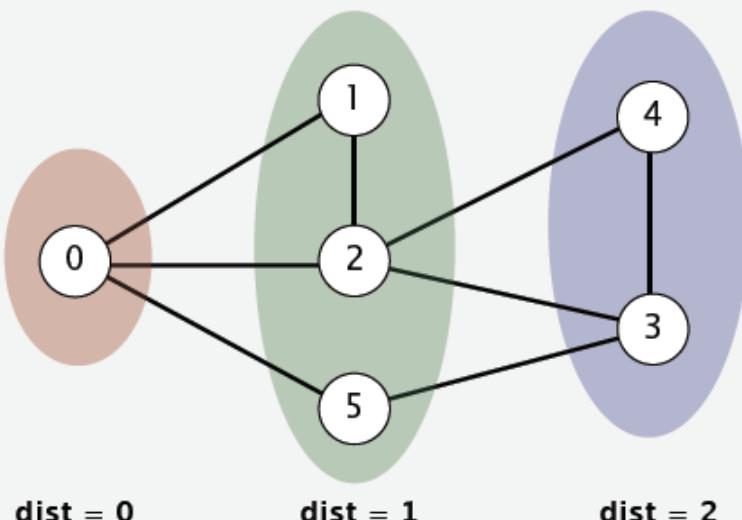
- Q. In which order does BFS examine vertices?
A. Increasing distance (number of edges) from s .

queue always consists of ≥ 0 vertices of distance k from s ,
followed by ≥ 0 vertices of distance $k+1$

Proposition. In any connected graph G , BFS computes shortest paths from s to all other vertices in time proportional to $E + V$.



graph G



dist = 0

dist = 1

dist = 2

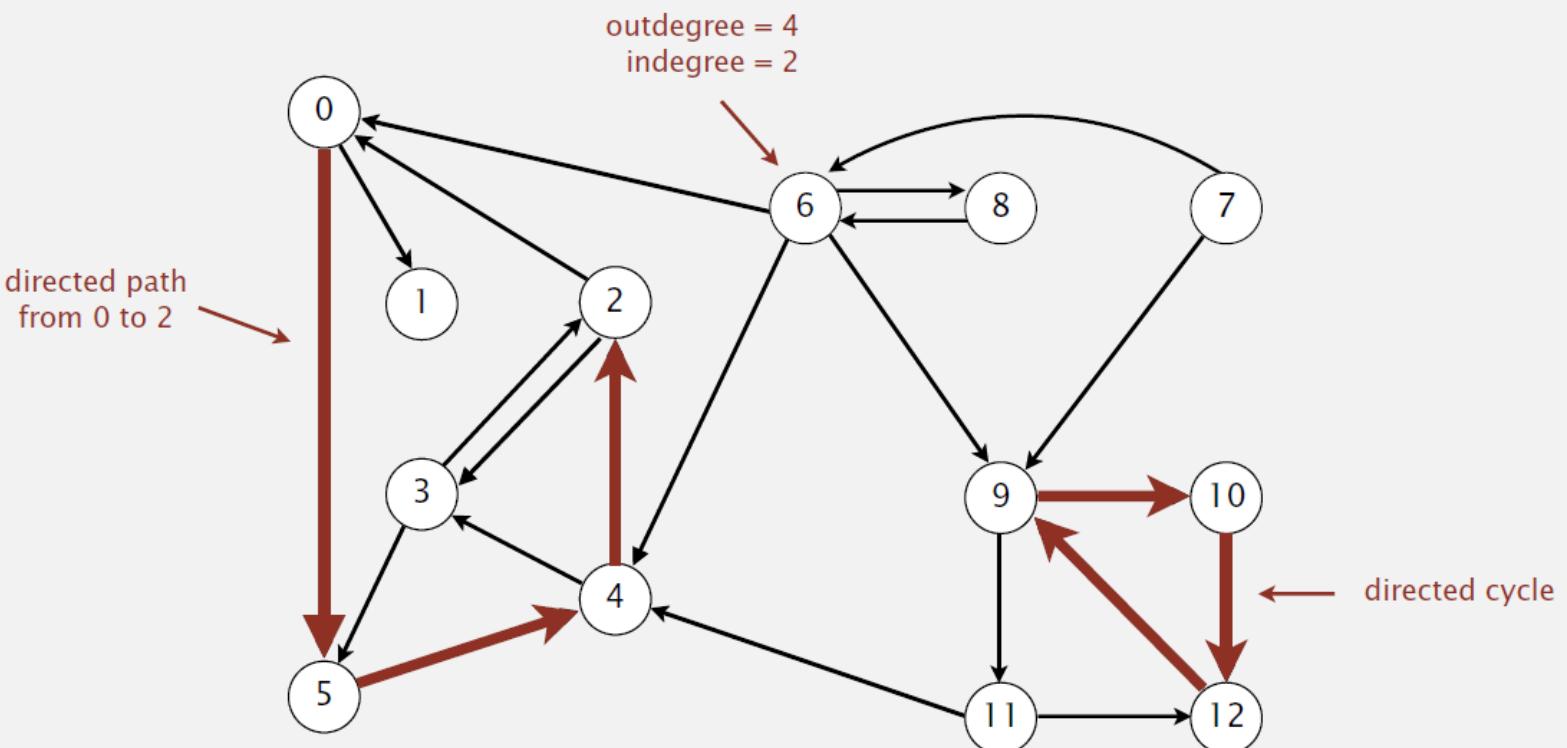
How to answer?

- › Does a path from A to B exist?
 - › Find a shortest path from A to B?
 - › Is A connected to all other nodes in the system?
-
- › Both DFS and BFS are brute-force/exhaustive-search algorithms
 - › Other algorithms to follow

Directed Graphs

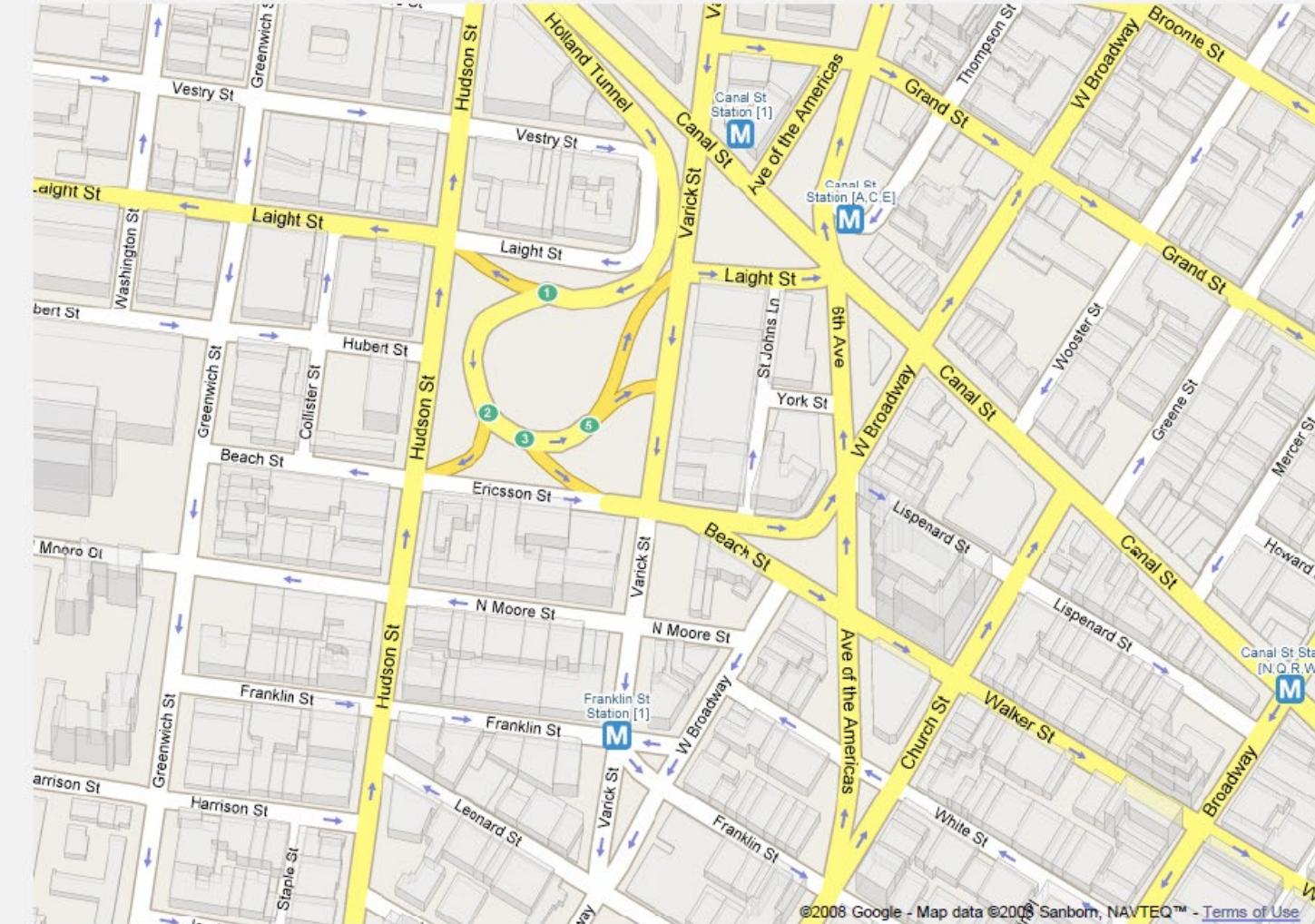
Directed graphs

Digraph. Set of vertices connected pairwise by directed edges.



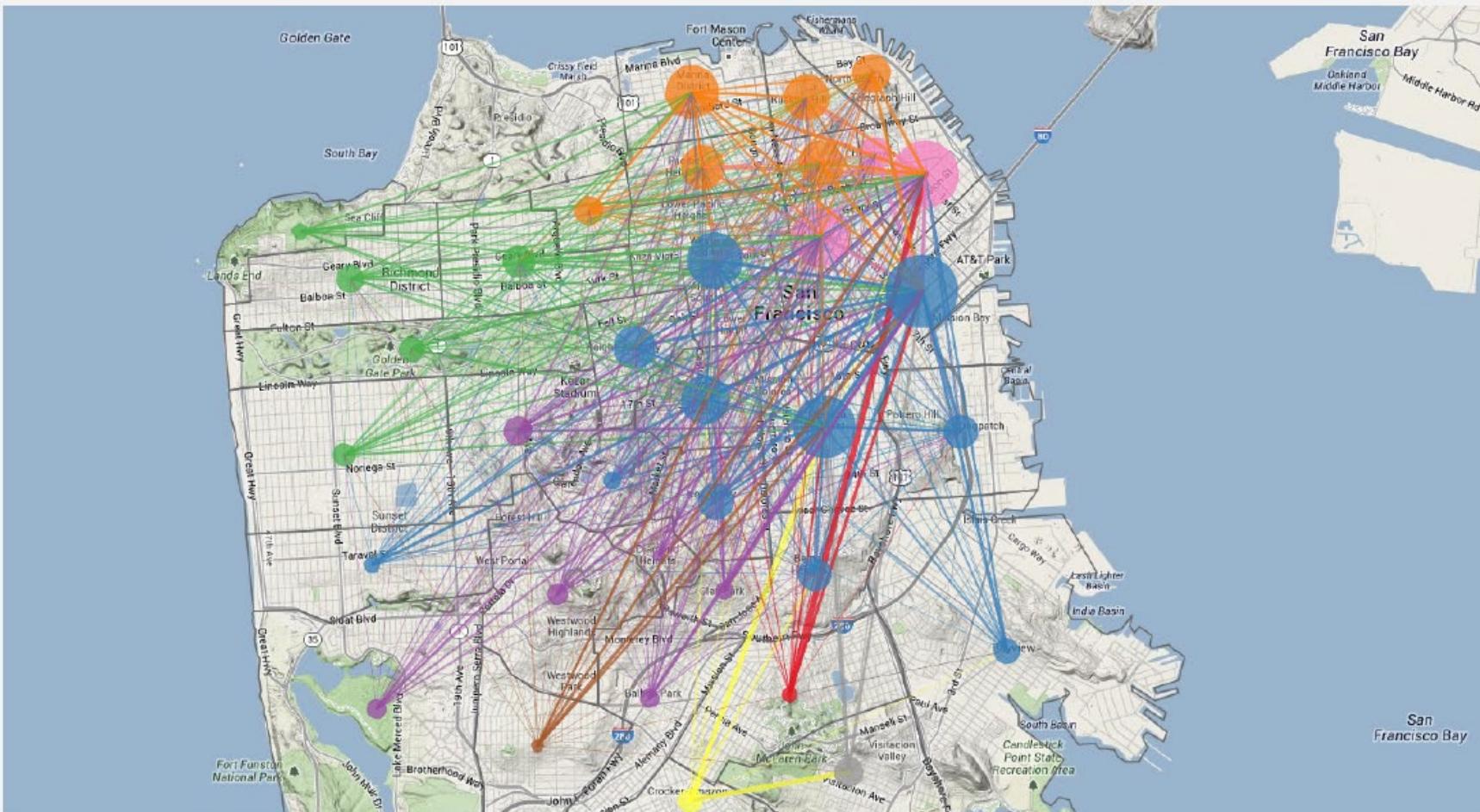
Road network

Vertex = intersection; edge = one-way street.



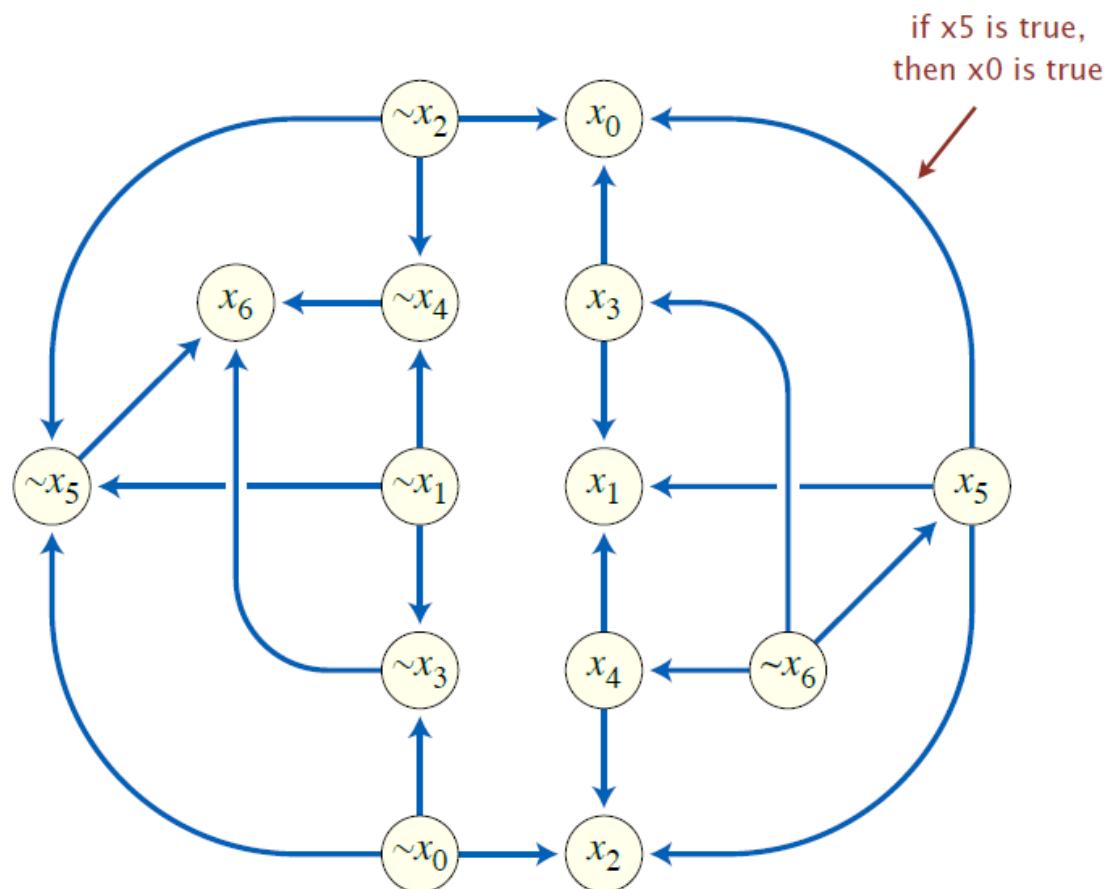
Uber taxi graph

Vertex = taxi pickup; edge = taxi ride.



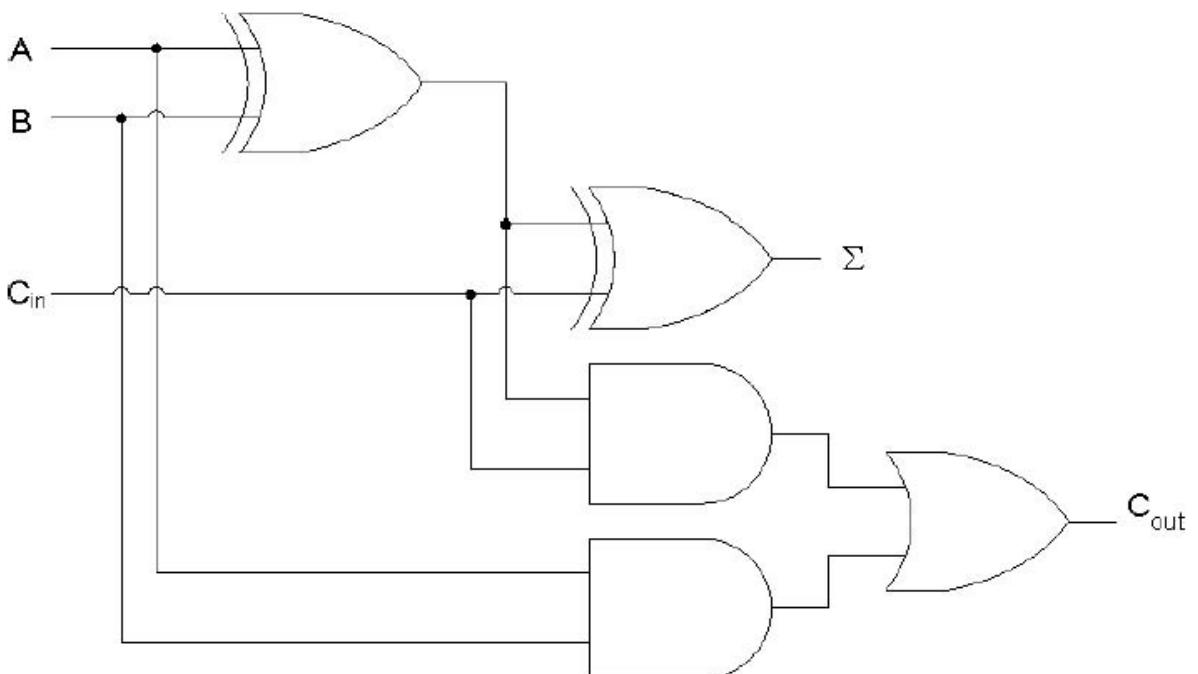
Implication graph

Vertex = variable; edge = logical implication.



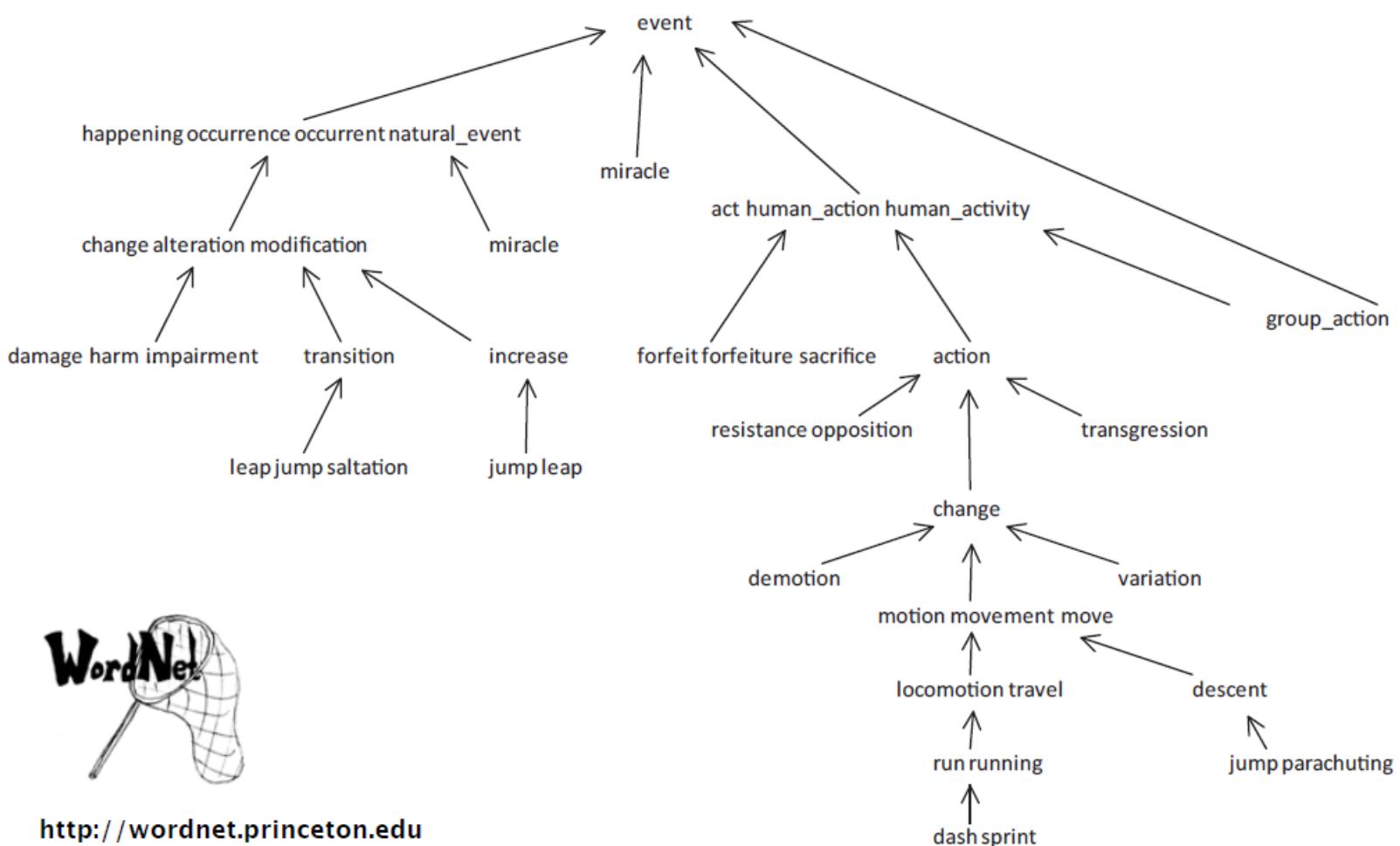
Combinational circuit

Vertex = logical gate; edge = wire.



WordNet graph

Vertex = synset; edge = hypernym relationship.



Digraph applications

digraph	vertex	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Digraph API

Almost identical to Graph API.

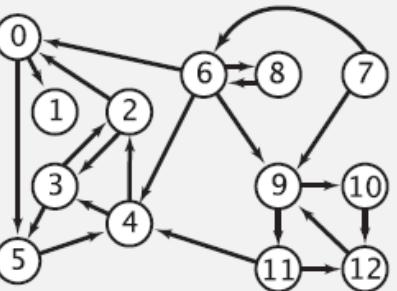
<code>public class Digraph</code>	
<code> Digraph(int V)</code>	<i>create an empty digraph with V vertices</i>
<code> Digraph(In in)</code>	<i>create a digraph from input stream</i>
<code> void addEdge(int v, int w)</code>	<i>add a directed edge $v \rightarrow w$</i>
<code> Iterable<Integer> adj(int v)</code>	<i>vertices pointing from v</i>
<code> int V()</code>	<i>number of vertices</i>
<code> int E()</code>	<i>number of edges</i>
<code> Digraph reverse()</code>	<i>reverse of this digraph</i>
<code> String toString()</code>	<i>string representation</i>

Digraph API

tinyDG.txt

V → 13
22 ← *E*

4 2
2 3
3 2
6 0
0 1
2 0
11 12
12 9
9 10
9 11
7 9
10 12
11 4
4 3
3 5
6 8
8 6
:



% java Digraph tinyDG.txt

0->5
0->1
2->0
2->3
3->5
3->2
4->3
4->2
5->4
:
11->4
11->12
12->9

```
In in = new In(args[0]);
```

```
Digraph G = new Digraph(in);
```

read digraph from
input stream

```
for (int v = 0; v < G.V(); v++)
```

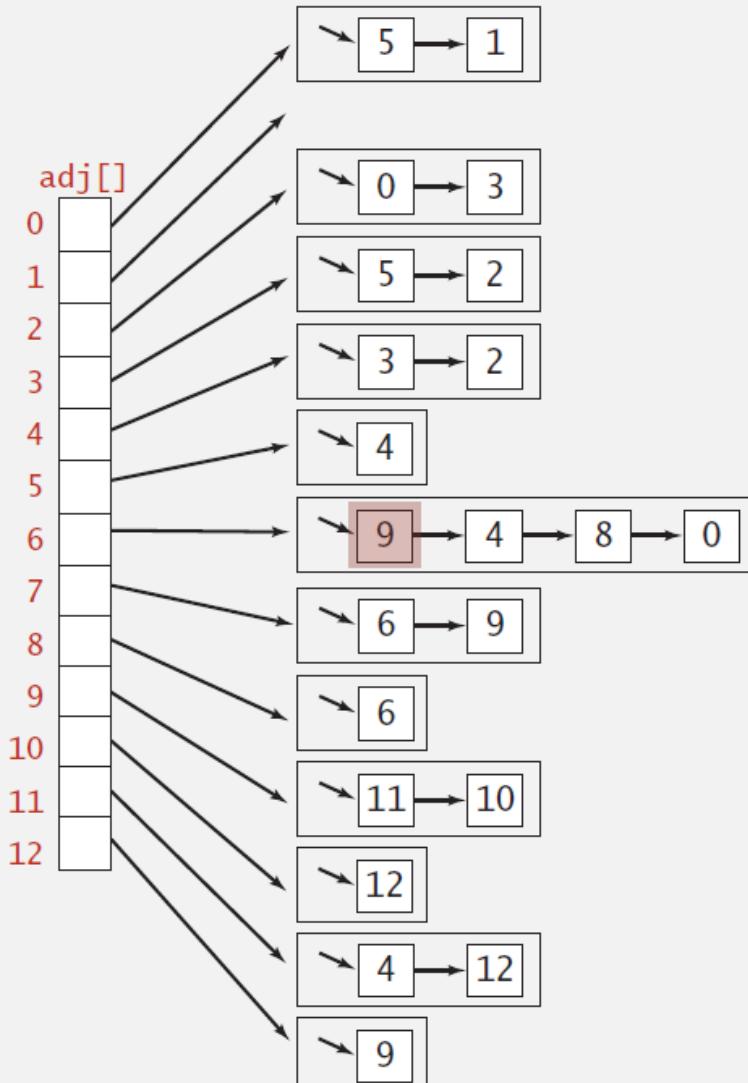
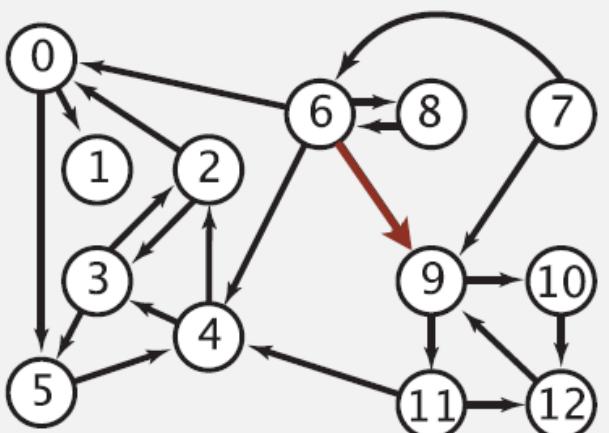
```
for (int w : G.adj(v))
```

```
StdOut.println(v + "->" + w);
```

print out each
edge (once)

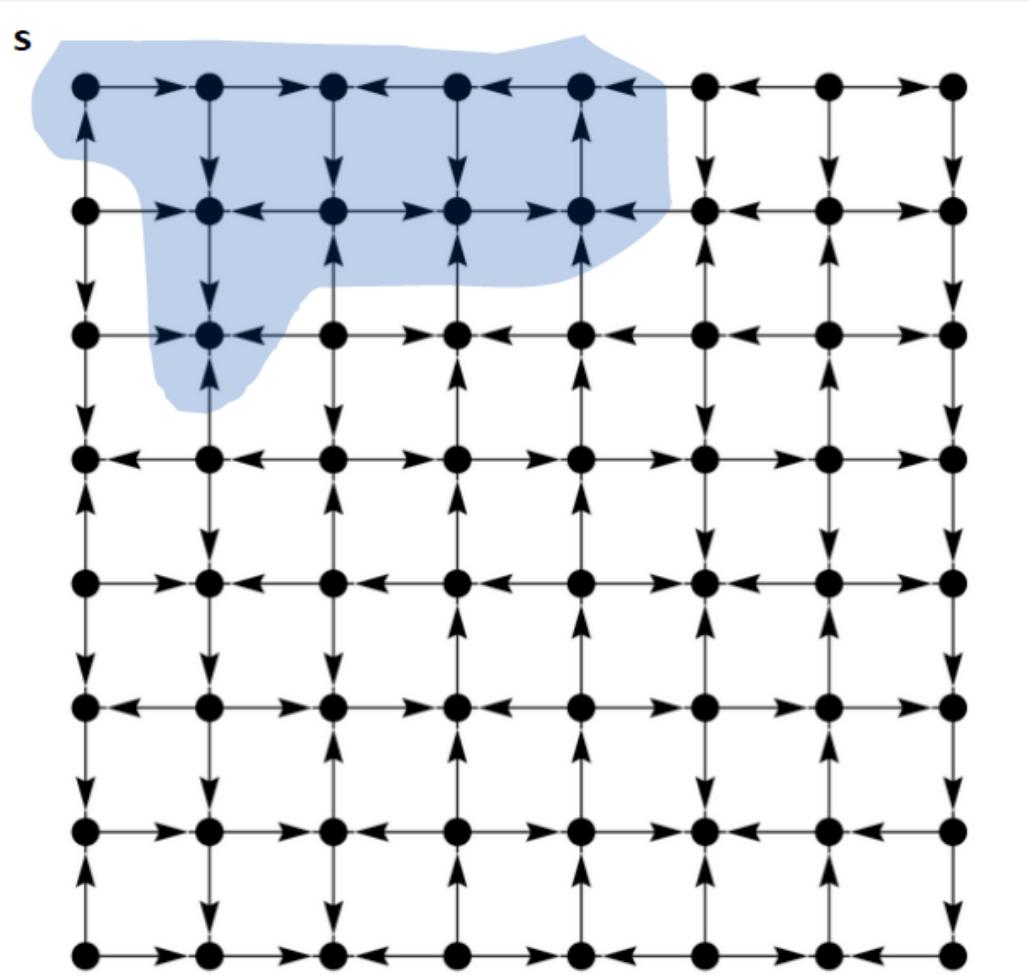
Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.



Reachability

Problem. Find all vertices reachable from s along a directed path.



Search in digraphs

- › BFS and DFS work

DFS in digraphs

DFS enables direct solution of simple digraph problems.

- ✓ • Reachability.
- Path finding.
- Topological sort.
- Directed cycle detection.

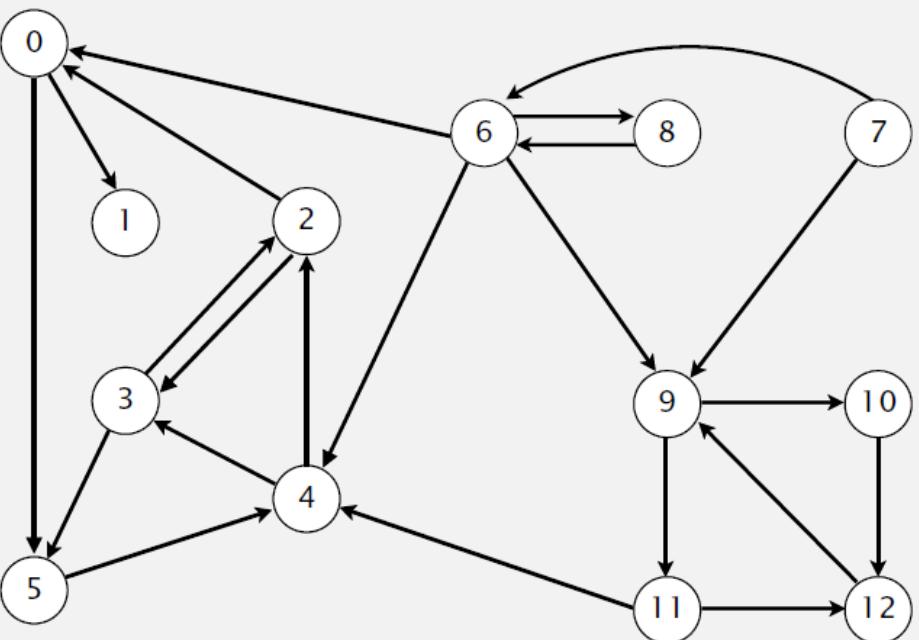
Basis for solving difficult digraph problems.

- 2-satisfiability.
- Directed Euler path.
- Strongly-connected components.

DFS demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v .

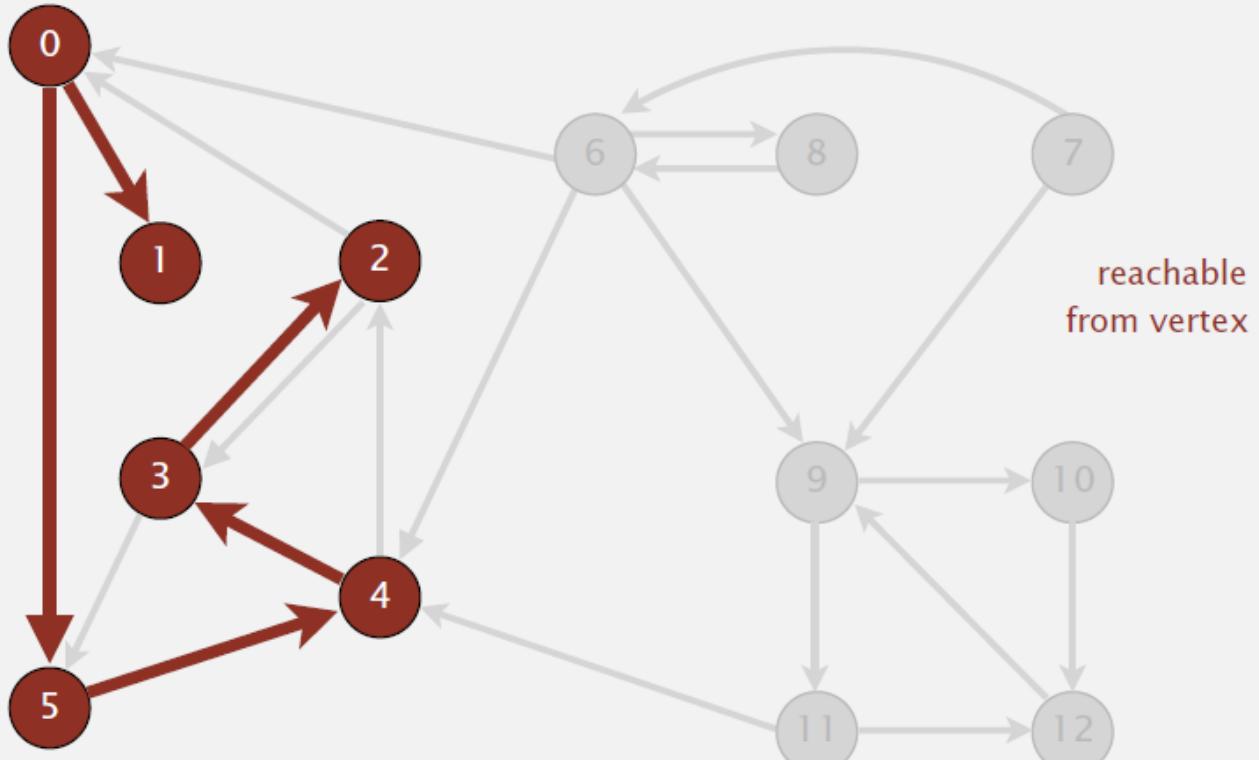


a directed graph

4→2
2→3
3→2
6→0
0→1
2→0
11→12
12→9
9→10
9→11
8→9
10→12
11→4
4→3
3→5
6→8
8→6
5→4
0→5
6→4
6→9
7→6

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v .



reachable from 0

v	marked[]	edgeTo[]
0	T	-
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a **digraph** algorithm.

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

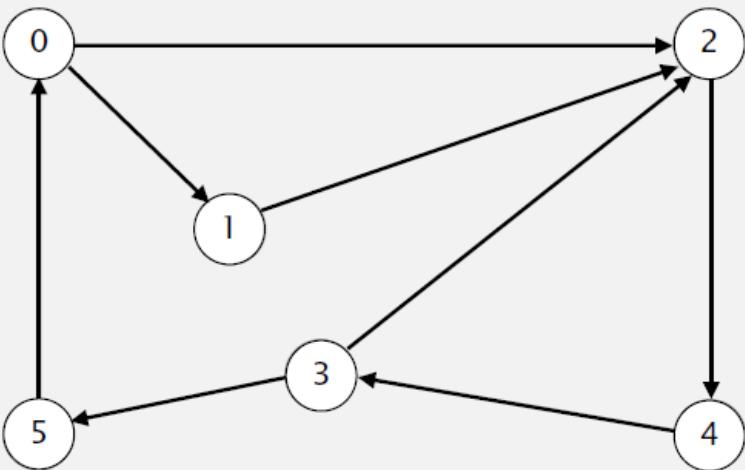
- remove the least recently added vertex v
- for each unmarked vertex pointing from v :
add to queue and mark as visited.

Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E + V$.

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



tinyDG2.txt

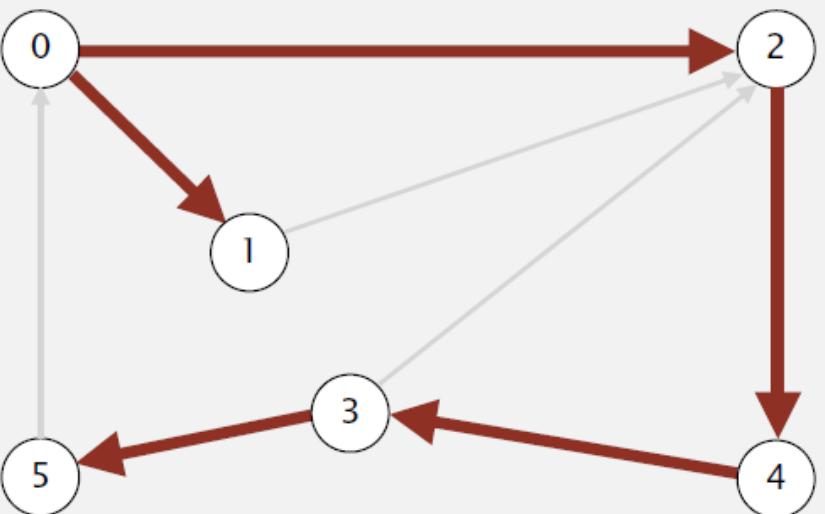
v → 6
E ← 8
5 0
2 4
3 2
1 2
0 1
4 3
3 5
0 2

graph G

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



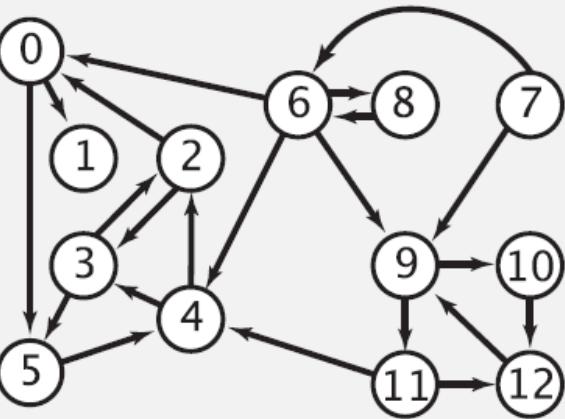
v	edgeTo[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

Multiple-source shortest paths

Multiple-source shortest paths. Given a digraph and a **set** of source vertices, find shortest path from any vertex in the set to each other vertex.

Ex. $S = \{ 1, 7, 10 \}$.

- Shortest path to 4 is $7 \rightarrow 6 \rightarrow 4$.
- Shortest path to 5 is $7 \rightarrow 6 \rightarrow 0 \rightarrow 5$.
- Shortest path to 12 is $10 \rightarrow 12$.
- ...



Q. How to implement multi-source shortest paths algorithm?

A. Use BFS, but initialize by enqueueing all source vertices.

Applications

- › AI – search, constraint programming, NLP
- › Topological sort
- › Directed cycle detection
- › Strongly-connected components

Topological sort

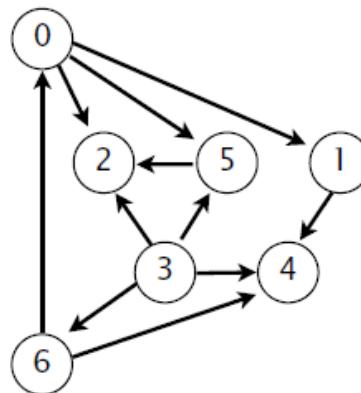
Precedence scheduling

Goal. Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

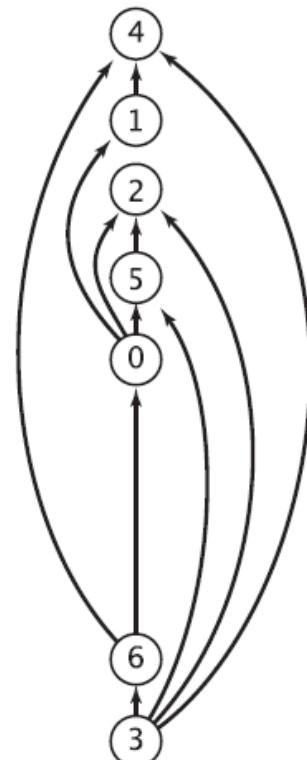
Digraph model. vertex = task; edge = precedence constraint.

- 0. Algorithms
- 1. Complexity Theory
- 2. Artificial Intelligence
- 3. Intro to CS
- 4. Cryptography
- 5. Scientific Computing
- 6. Advanced Programming

tasks



precedence constraint graph



feasible schedule

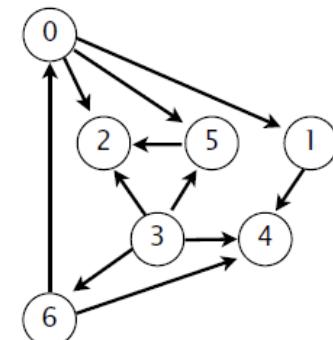
Topological sort

DAG. Directed **acyclic** graph.

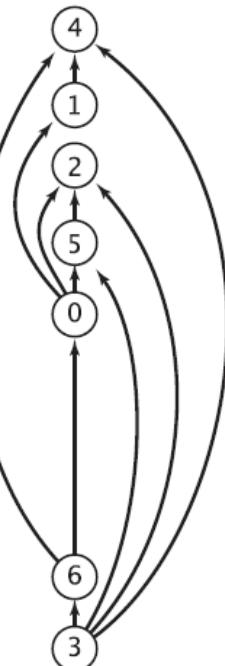
Topological sort. Redraw DAG so all edges point upwards.

$0 \rightarrow 5$	$0 \rightarrow 2$
$0 \rightarrow 1$	$3 \rightarrow 6$
$3 \rightarrow 5$	$3 \rightarrow 4$
$5 \rightarrow 2$	$6 \rightarrow 4$
$6 \rightarrow 0$	$3 \rightarrow 2$
$1 \rightarrow 4$	

directed edges



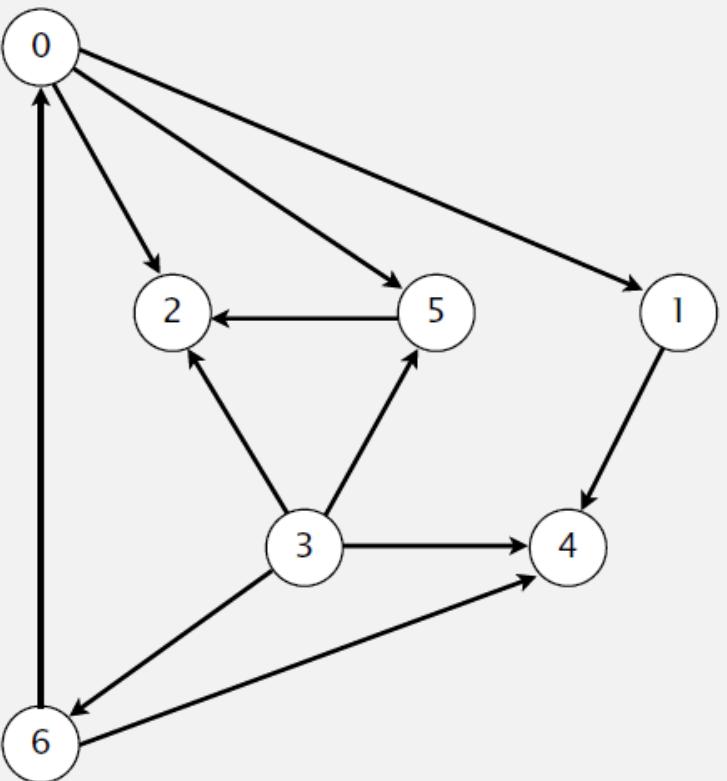
DAG



topological order

Solution. DFS. What else?

- Run depth-first search.
- Return vertices in reverse postorder.



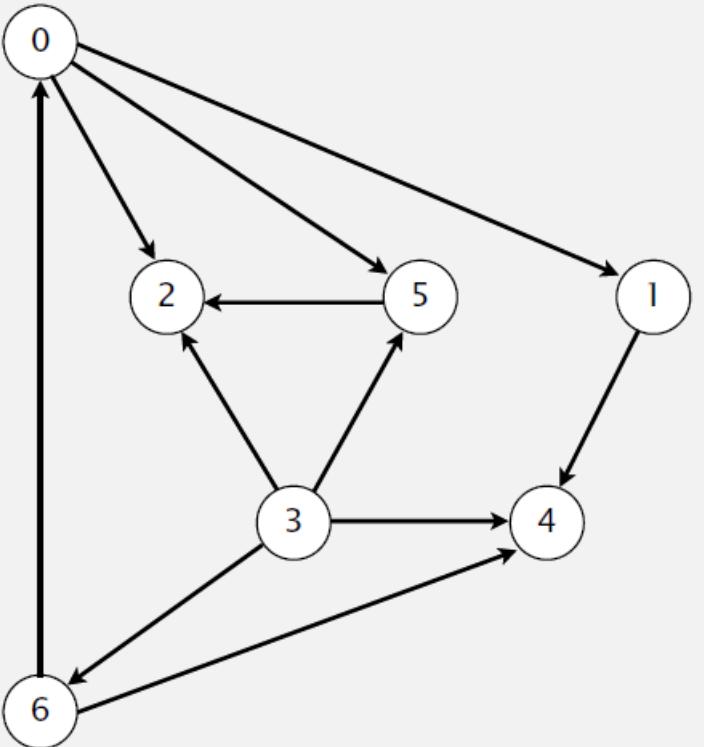
tinyDAG7.txt

7
11
0 5
0 2
0 1
3 6
3 5
3 4
5 2
6 4
6 0
3 2

a directed acyclic graph

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



postorder

4 1 2 5 0 6 3

topological order

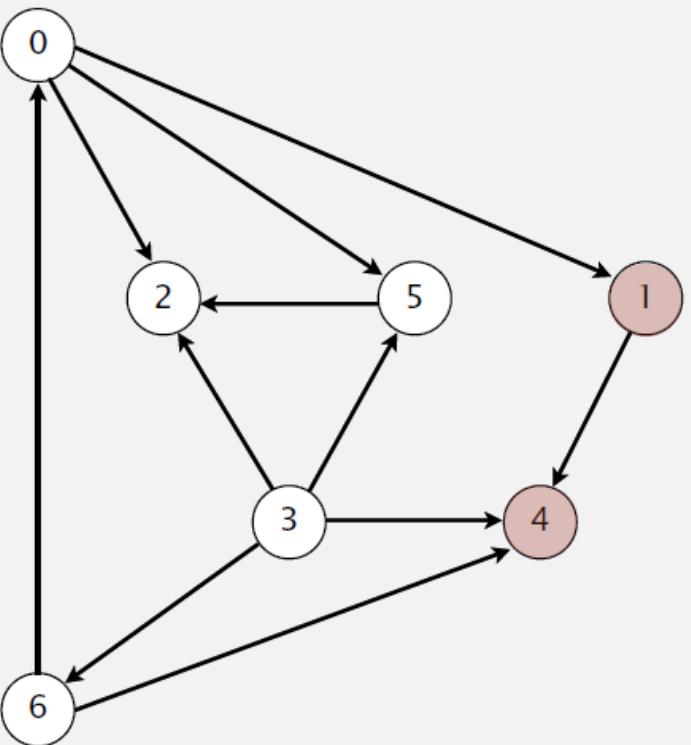
3 6 0 5 2 1 4

done

Topological sort in a DAG: intuition

Why does topological sort algorithm work?

- First vertex in postorder has outdegree 0.
- Second-to-last vertex in postorder can only point to last vertex.
- ...



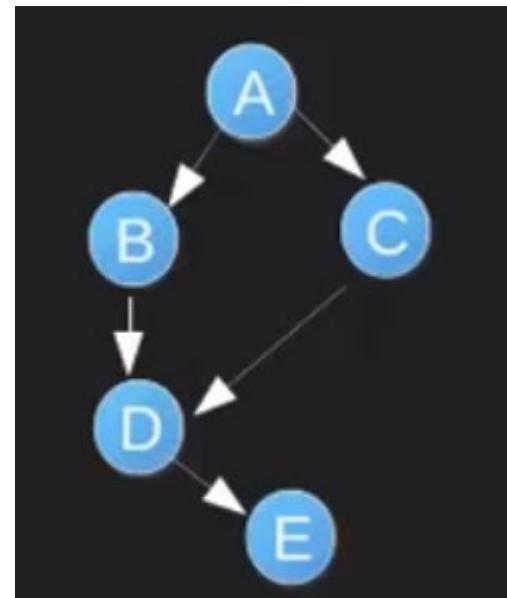
postorder

4 1 2 5 0 6 3

topological order

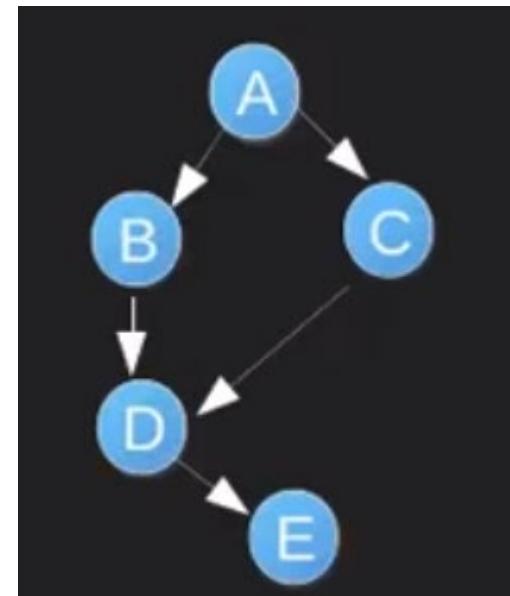
3 6 0 5 2 1 4

Topological sort



Topological sort

- › Not unique ordering
 - › A B C D E
 - › A C B D E
- › Complexity
 - DFS with an extra stack
 - Same as DFS which is $O(V+E)$



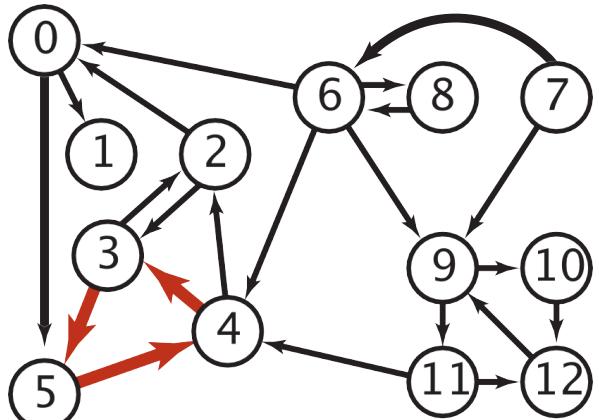
Directed Cycle Detection

Directed cycle detection

Proposition. A digraph has a topological order iff no directed cycle.

Pf.

- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.



a digraph with a directed cycle

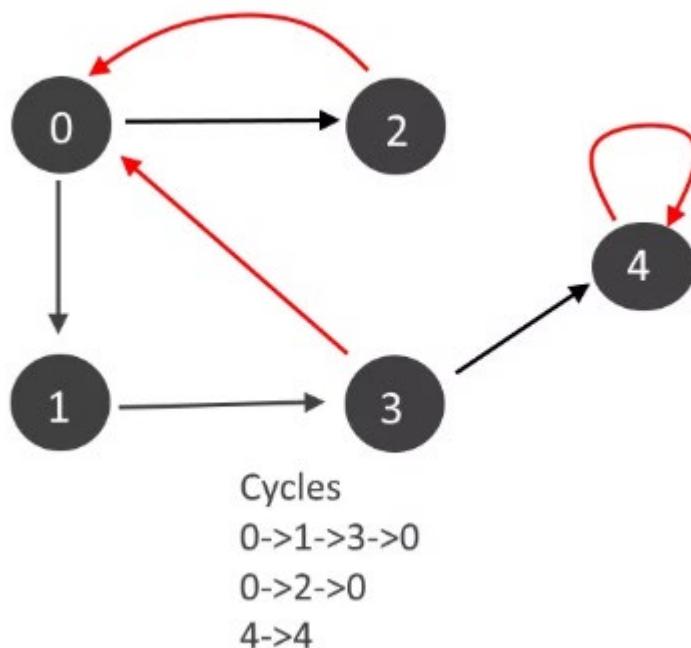
Goal. Given a digraph, find a directed cycle.

Directed cycle detection

Graph contains cycle if there are any back edges.

1. Edge from a vertex to itself. Self loop.
2. Edge from any descendent back to vertex.

- Do the DFS from each vertex
- For DFS from each vertex, keep track of visiting vertices in a recursion stack (array).
- If you encounter a vertex which already present in recursion stack then we have found a cycle.
- Use visited[] for DFS to keep track of already visited vertices.



Directed cycle detection application: precedence scheduling

Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

PAGE 3	DEPARTMENT	COURSE	DESCRIPTION	PREREQS
	COMPUTER SCIENCE	CPSC 432	INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.	CPSC 432

<http://xkcd.com/754>

Remark. A directed cycle implies scheduling problem is infeasible.

Directed cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

```
public class A extends B
{
    ...
}
```

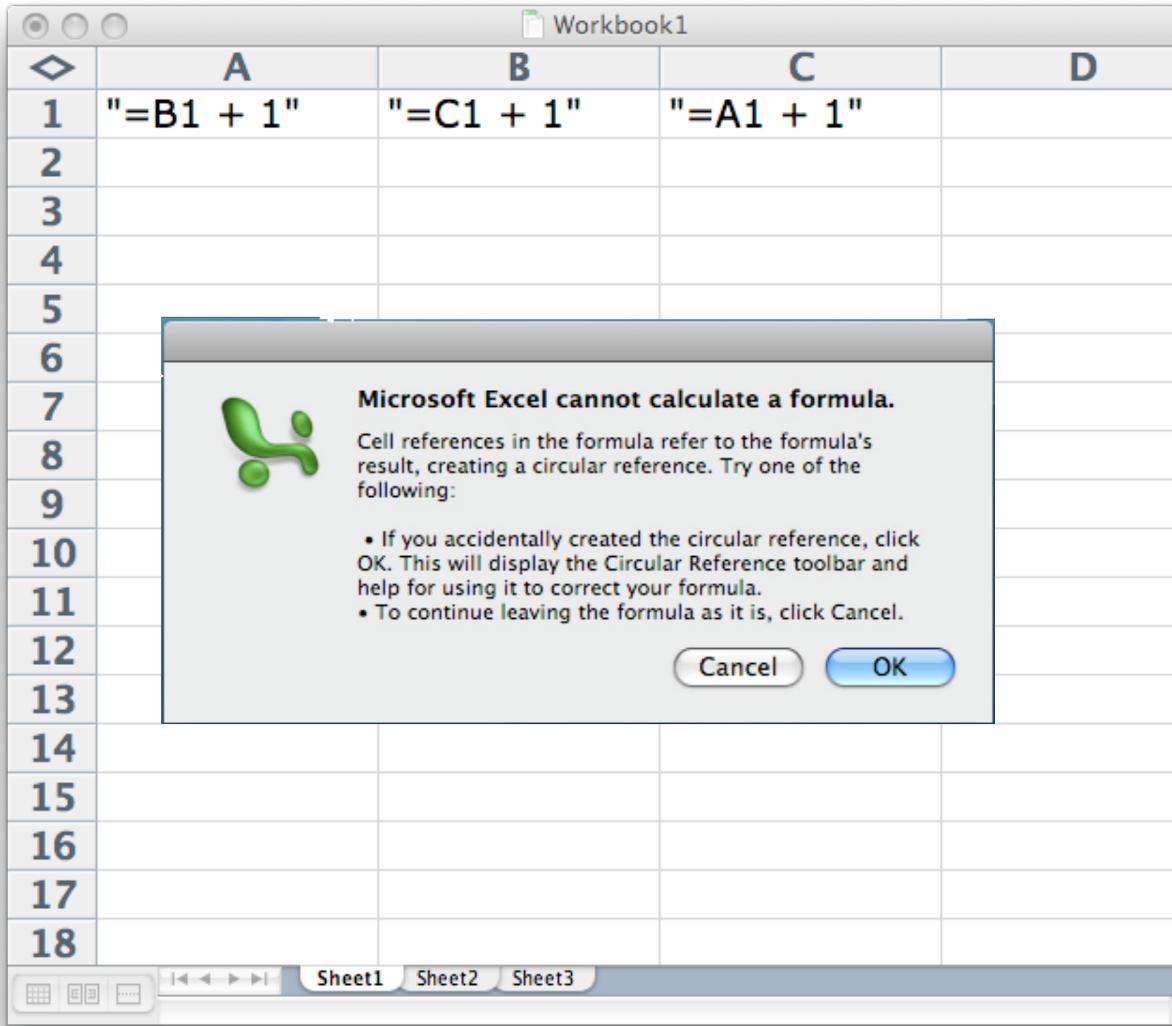
```
public class B extends C
{
    ...
}
```

```
public class C extends A
{
    ...
}
```

```
% javac A.java
A.java:1: cyclic inheritance
involving A
public class A extends B { }
^
1 error
```

Directed cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection (and has a circular reference toolbar!)



Depth-first search orders

Observation. DFS visits each vertex exactly once. The order in which it does so can be important.

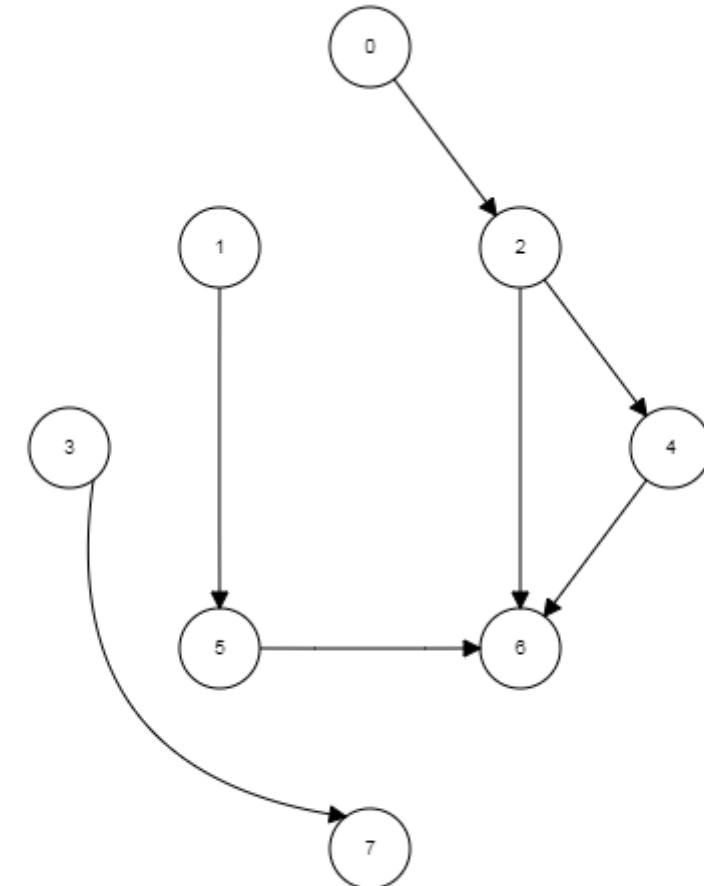
Orderings.

- Preorder: order in which `dfs()` is called.
- Postorder: order in which `dfs()` returns.
- Reverse postorder: reverse order in which `dfs()` returns.

```
private void dfs(Graph G, int v)
{
    marked[v] = true;
    preorder.enqueue(v);
    for (int w : G.adj(v))
        if (!marked[w]) dfs(G, w);
    postorder.enqueue(v);
    reversePostorder.push(v);
}
```

Topological sort exercise

- Show the nodes in the graph in topological order
- Keep track of the stack of nodes as they are added and removed
- Is the ordering unique? If not, show another valid topological ordering



Edit Mode is: ON [?](#)

Tests, Surveys and Pools

CSU22010-A-YEAR02-201920 ALGORITHMS AND DATA STRUCTURES

Tests Test Canvas : Last year's e-test for practice

Test Canvas: Last year's e-test for practice*The Test Canvas lets you add, edit and reorder questions, as well as review a test.* [More Help](#)[Create Question](#)[Reuse Question](#)[Upload Questions](#)[Question Settings](#)**Description**

Instructions Please do not start the test until instructed to do so by a demonstrator.

The test is multiple choice, and it consists of 14 questions. You have 45 minutes in total to complete the test.

Once you press submit your answers cannot be changed, so please double-check all of your answers before submitting.

Total Questions

Total Points 100

Select: All None Select by Type: - Question Type - ▾[Delete](#)[Points](#)[Update](#)[Hide Question Details](#)Points: **6****1. Multiple Choice: Insertion sort is an example of which...**

Question	Insertion sort is an example of which algorithm design approach
-----------------	---

Answer	Greedy
---------------	--------

Brute force

<input checked="" type="checkbox"/> Decrease and conquer
--

Divide and conquer



Points: 6

2. Multiple Choice: Which one of the sorting algorithms I...

Question Which one of the sorting algorithms listed below is NOT stable?

Answer Insertion

Selection

Merge sort

All 3 are stable



Points: 6

3. Multiple Choice: What is the worst case input for stan...

Question What is the worst case input for standard quicksort, if the first element in the (sub)array is always used as a pivot?

Answer already sorted array

random array

array with a very few unique values

it does not matter - performance of quicksort is the same regardless of input order



Points: 6

4. Multiple Choice: What is the worst case complexity for...

Question What is the worst case complexity for merge sort algorithm?

Answer $O(n^2)$

$O(n \log n)$

$O(n)$

$O(\log n)$

Points: **6**

5. Multiple Choice: Which of the following algorithms,&nb...

Question Which of the following algorithms, in its basic implementation, has space complexity of $O(n)$?

Answer insertion sort

bubble sort

quick sort

merge sort

Points: **6**

6. Multiple Choice: Given an array of 8 almost sorted int...

Question Given an array of 8 almost sorted integers, which of the following algorithms would be the most suitable to use to sort it?

Answer Insertion sort

Merge sort

Quick sort

Bubble sort



Points: 6

7. Multiple Choice: Which one of the following sort algor...**Question**

Which one of the following sort algorithms is generally not implemented using recursion?

Answer

Most Significant Digit Radix Sort

Least Significant Digit Radix Sort

Merge sort

quick sort



Points: 6

8. Multiple Choice: Which of the following substring sear...**Question**

Which of the following substring search algorithms does NOT require backup?

Answer

Knuth-Morris-Prath

Brute force approach

Boyer-Moore

all of the above require backup



Points: 6

9. Multiple Choice: Implementation of Knuth-Morris-Prath&...**Question**

Implementation of Knuth-Morris-Prath (KMP) string search algorithm requires the following amount of additional space, where M is the length of the string to be searched for, N is the length of the text in which we are searching, and R is the radix of the alphabet we are working with

Answer

N

 MR

R

M²

Points: 6

10. Multiple Choice: What is the worst case performance of...**Question**

What is the worst case performance of Boyer Moore string search algorithm, if N is the size of the string we are searching for, M is the size of the text in which we are searching, and R the radix of the algo

Answer

MR

M²

M

 MN

Points: 10

11. Multiple Choice: Assume you are searching for a string...**Question**

Assume you are searching for a string "BANANA" using Boyer-Moore algorithm. What is the final content of the "skip array" going to be, assuming the alphabet consists only of letters A B C D and N

Answer

A	1
B	0
C	0
D	0
N	2

A	1
B	0
C	-1
D	-1
N	2

A 5
B 0
C -1
D -1
N 4

A	5
B	0
C	-1
D	-1
N	-1

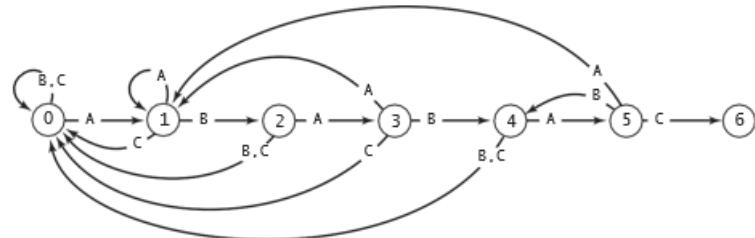
Points: 10

12. Multiple Choice: Assume the following DFA table has be...

Question

Assume the following DFA table has been constructed to search for a string "ABABAC" using KMP algorithm.

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
	A	1	1	3	1	5
dfa[][][j]	B	0	2	0	4	0
	C	0	0	0	0	6



The string which you are searching in is "ABABABABACAB".

Which of the answers below shows the correct trace of states of DFA this search would produce until it finds a full match?

Answer 0,1,2,3,4,5,0,1,2,3,4,5,4,5,6

0,1,2,3,4,5,4,5,4,5,6,4,5

0,1,2,3,4,5,4,5,4,5,6

0,1,2,3,4,5,6

Points: **10**

13. Multiple Choice: Assume you are required to sort the f...

Question

Assume you are required to sort the following array using the LSD (Least Significant Digit) radix sort algorithm.

C	A	T
C	A	R
A	R	T
B	A	R
B	O	T
C	O	T

Which of the answers below shows how will array look like after the FIRST pass (out of the total of 3 passes) of the algorithm through index key counting?

Answer



C	A	R
B	A	R
C	A	T
A	R	T
B	O	T
C	O	T

A	R	T
B	A	R
B	O	T
C	O	T
C	A	T
C	A	R

C	A	R
B	A	R
A	R	T
B	O	T
C	A	T
C	O	T

A	R	T
B	A	R
B	O	T
C	A	T
C	A	R
C	O	T

Points: **10****14. Multiple Choice: You are required to sort the followin...****Question**

You are required to sort the following array using merge sort { 2, 11, 7, 8, 3, 19, 13, 5 }. How many calls to the "merge" method will be made during the top down merge sort execution?

Answer

5

 7

8

10

Select:

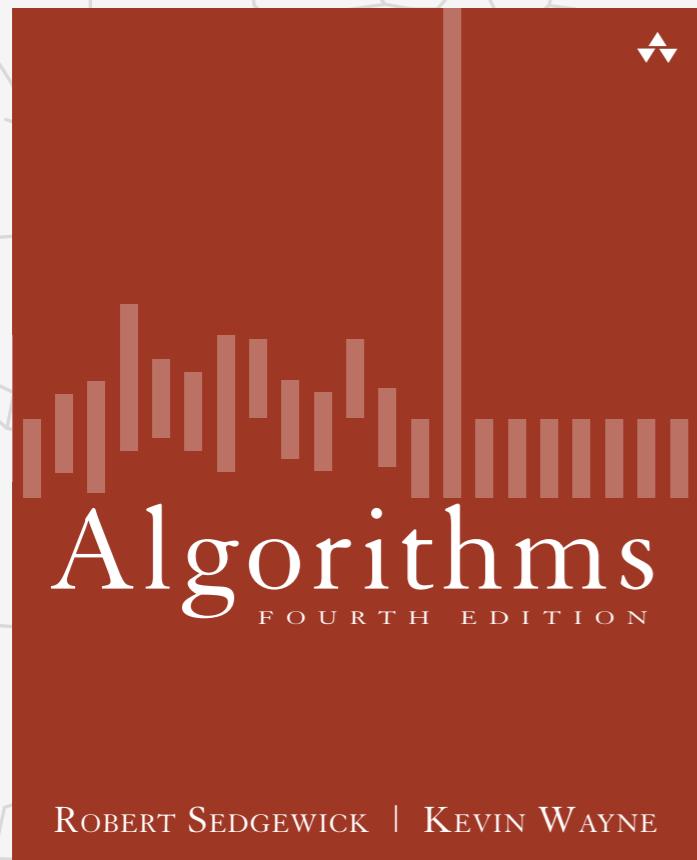
Select by Type:

- Question Type - ▼

← OK

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

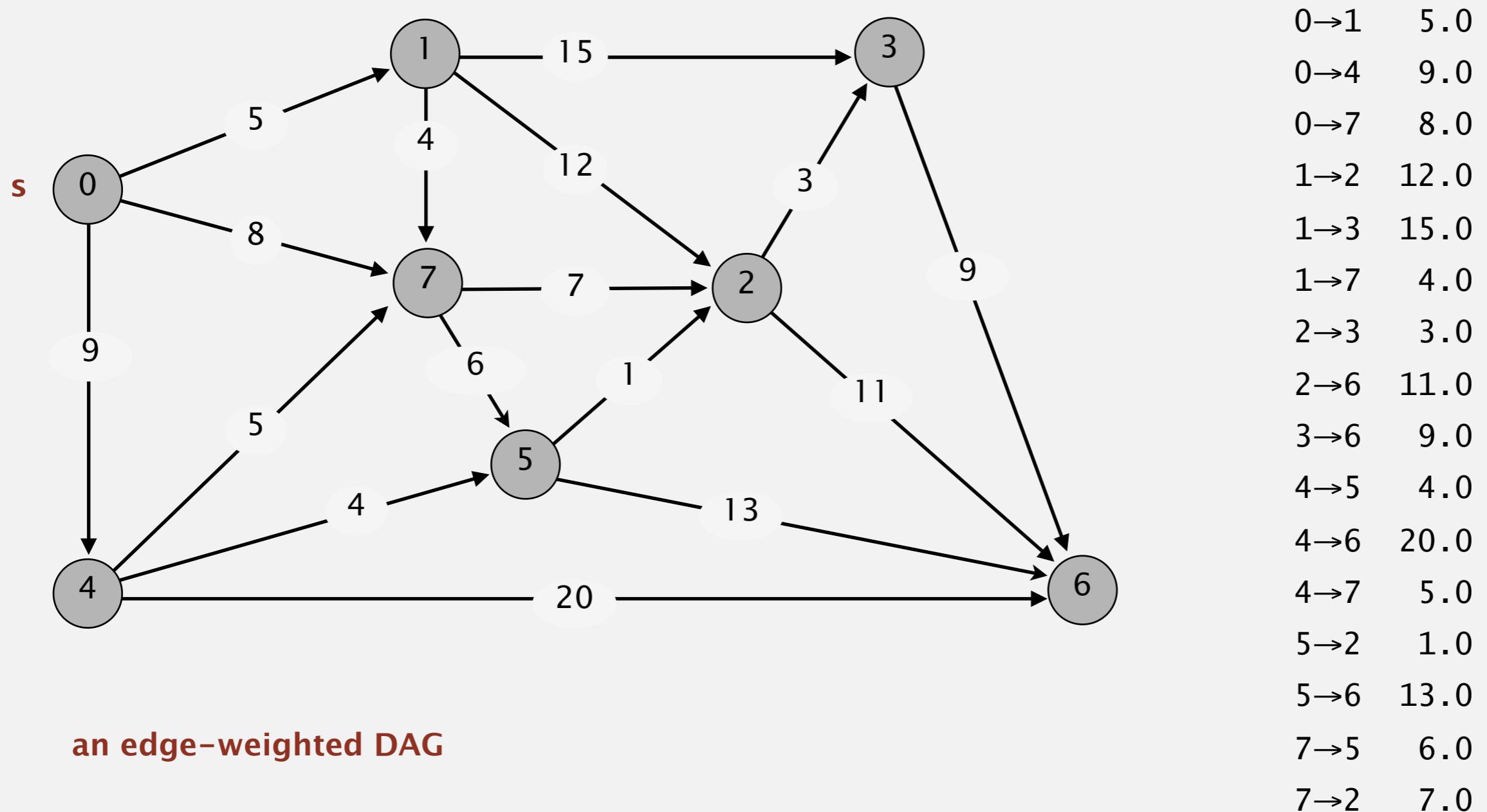


ACYCLIC SHORTEST PATHS DEMO

<http://algs4.cs.princeton.edu>

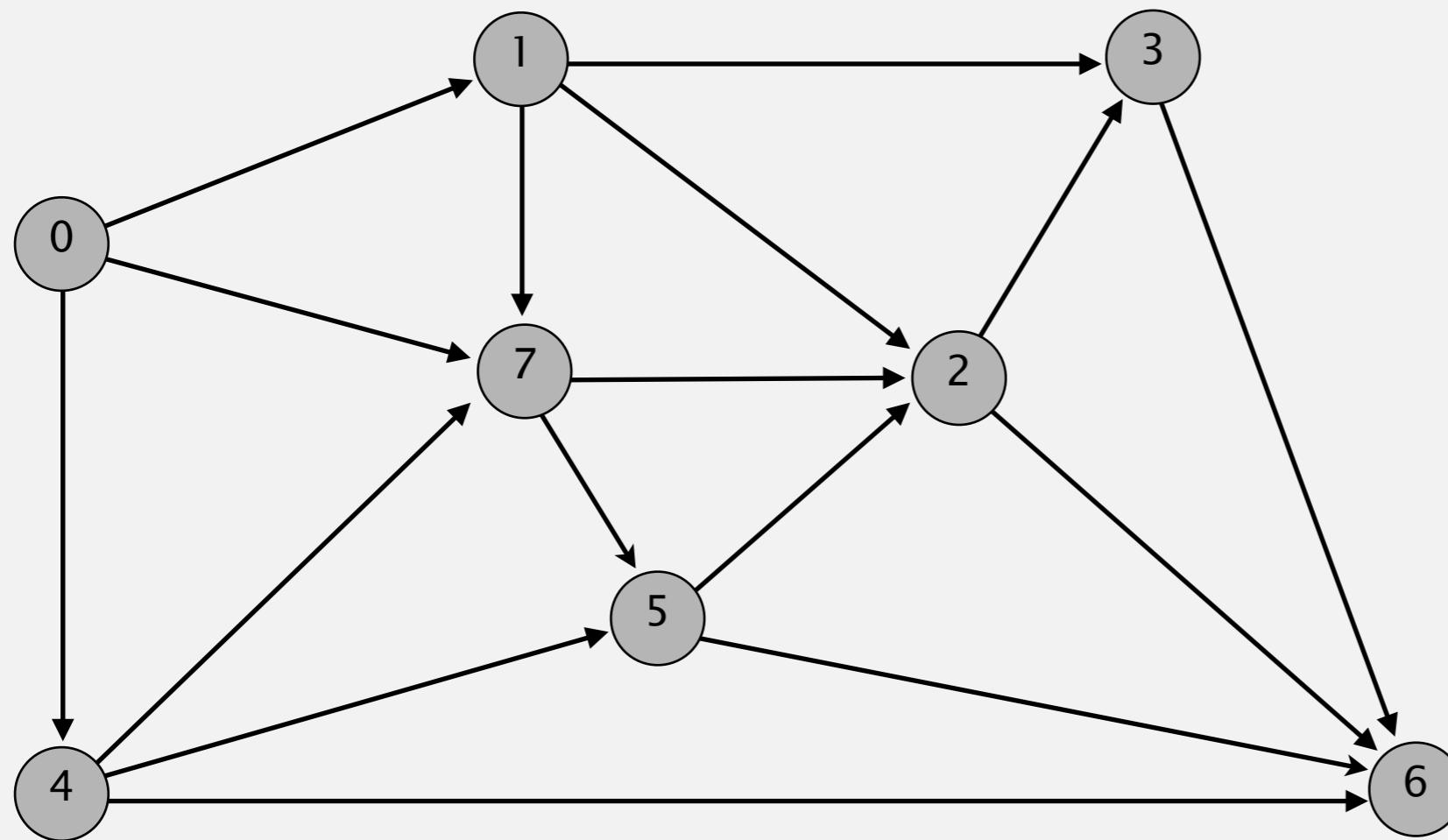
Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



Acyclic shortest paths demo

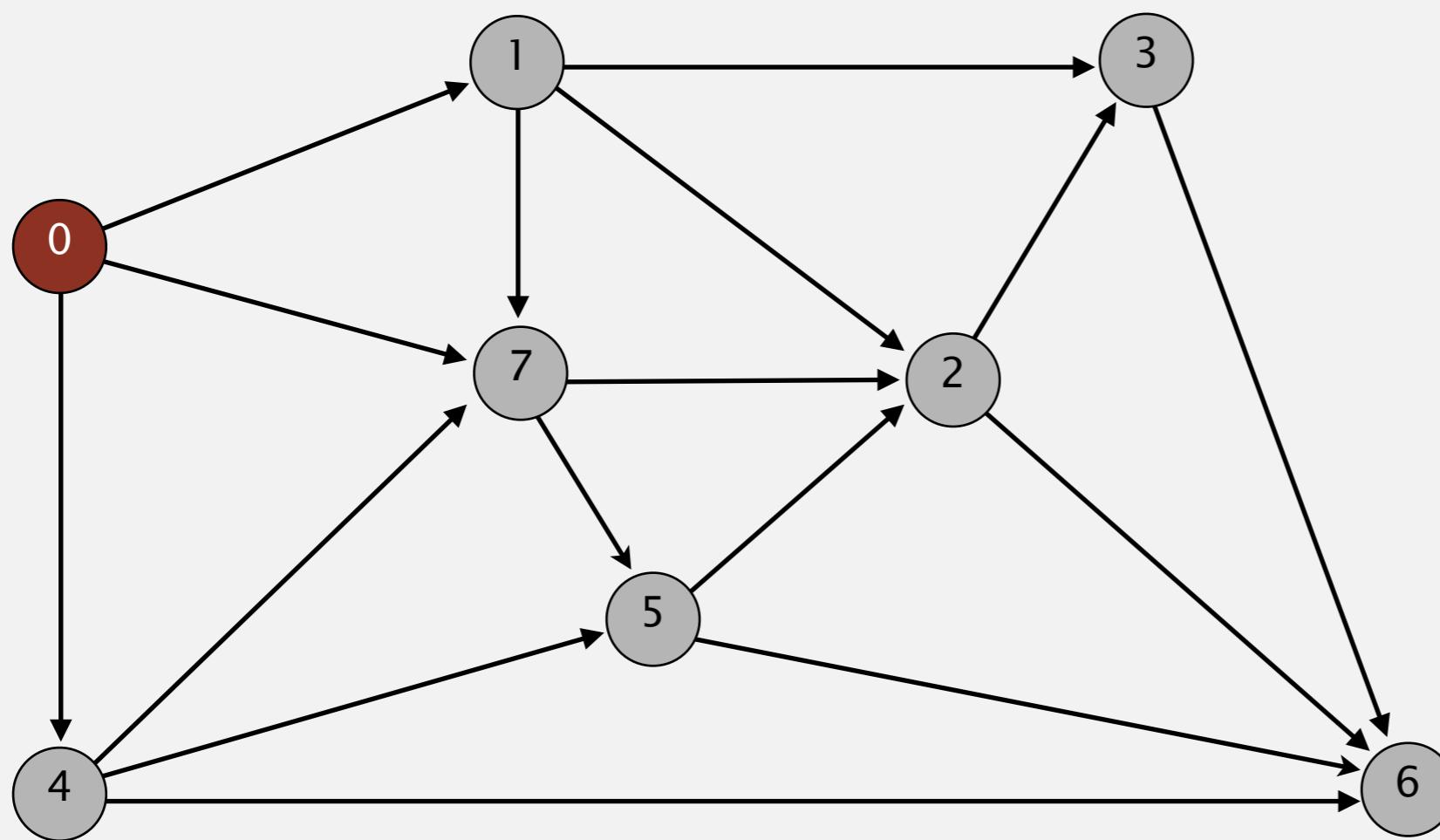
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



topological order: 0 1 4 7 5 2 3 6

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

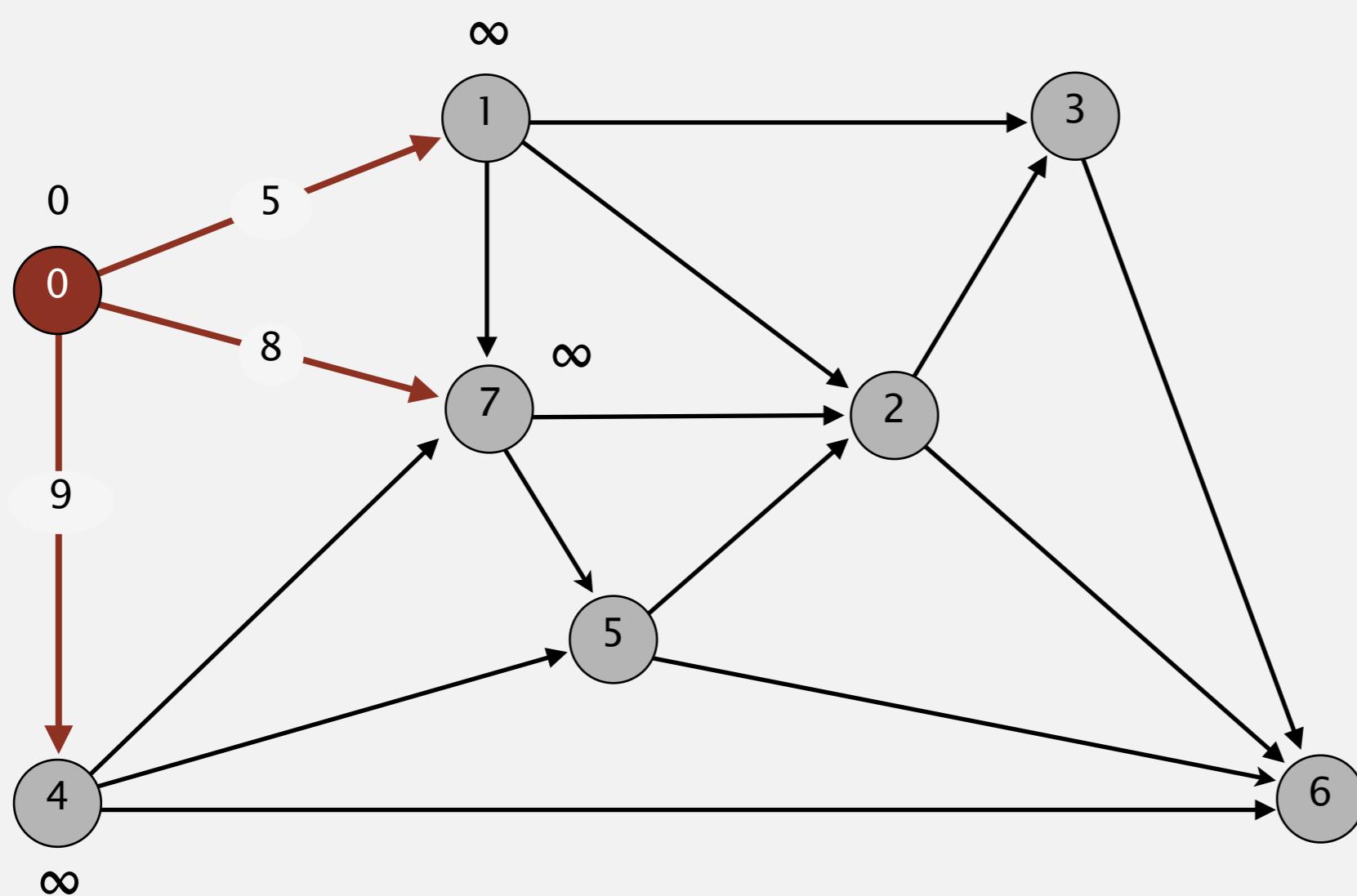


v	distTo[]	edgeTo[]
0	0.0	-
1		
2		
3		
4		
5		
6		
7		

choose vertex 0

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

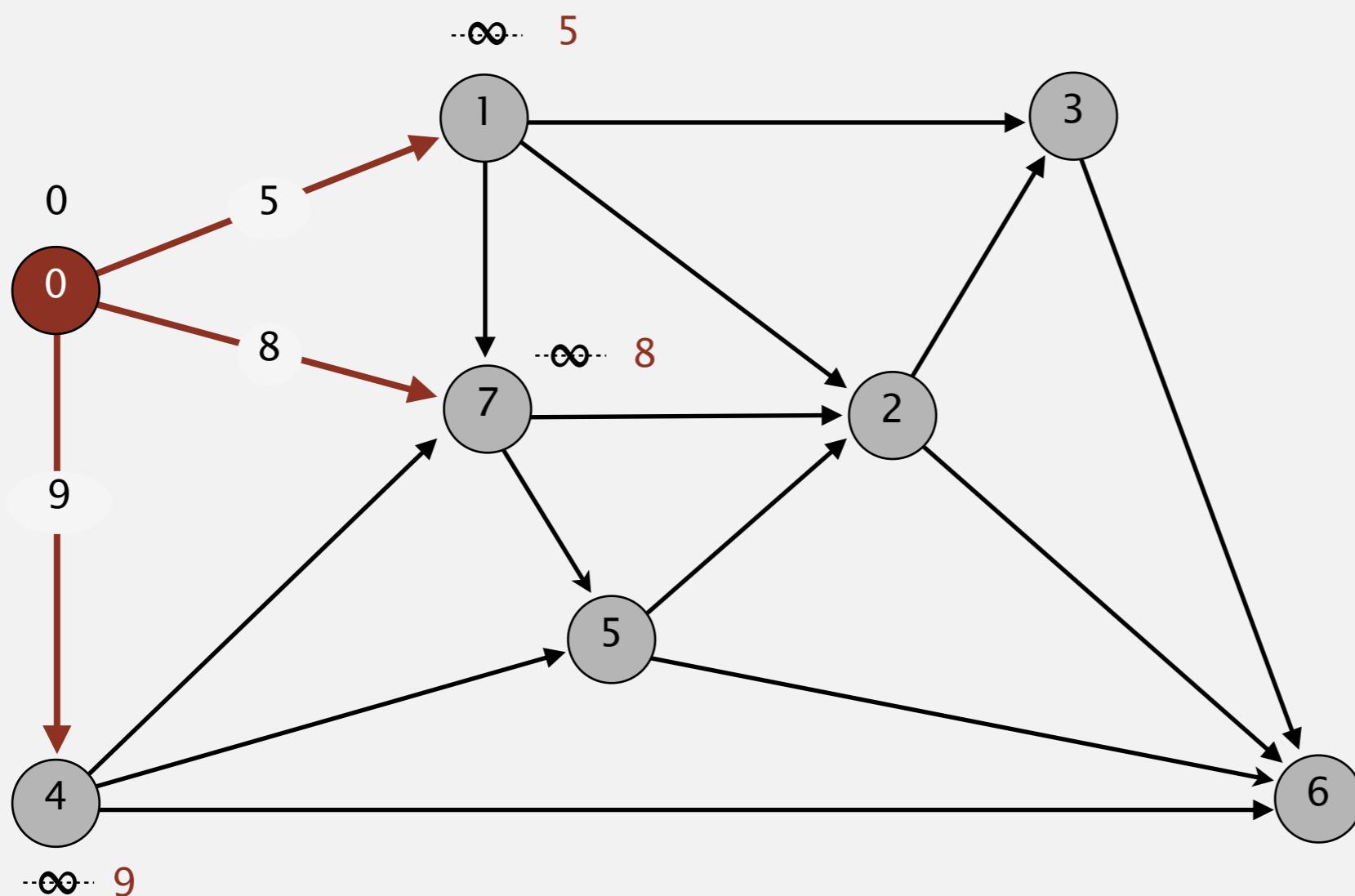


v	distTo[]	edgeTo[]
0	0.0	-
1	1	-
2	2	-
3	3	-
4	4	-
5	5	-
6	6	-
7	7	-

relax all edges pointing from 0

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

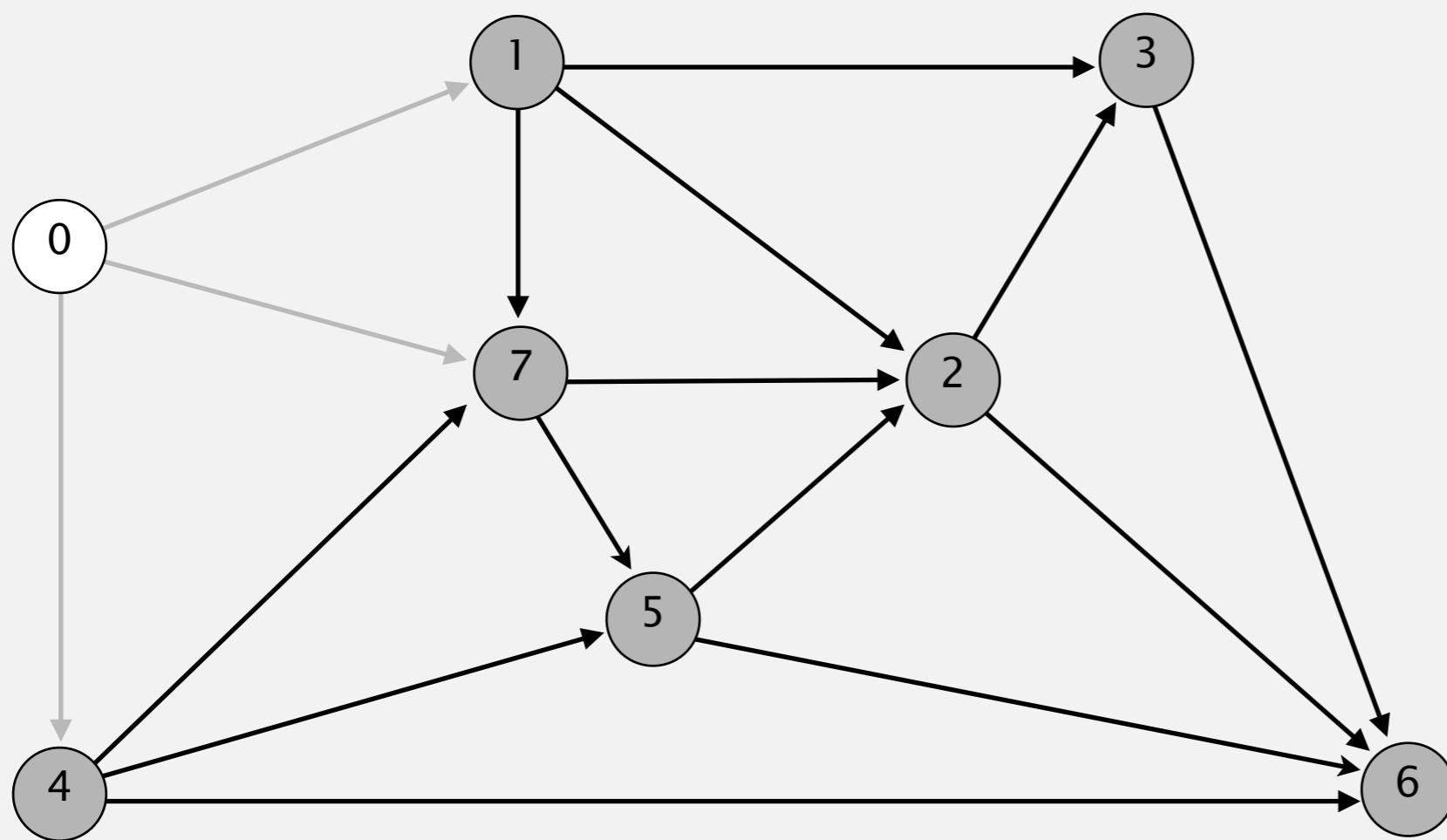


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

relax all edges pointing from 0

Acyclic shortest paths demo

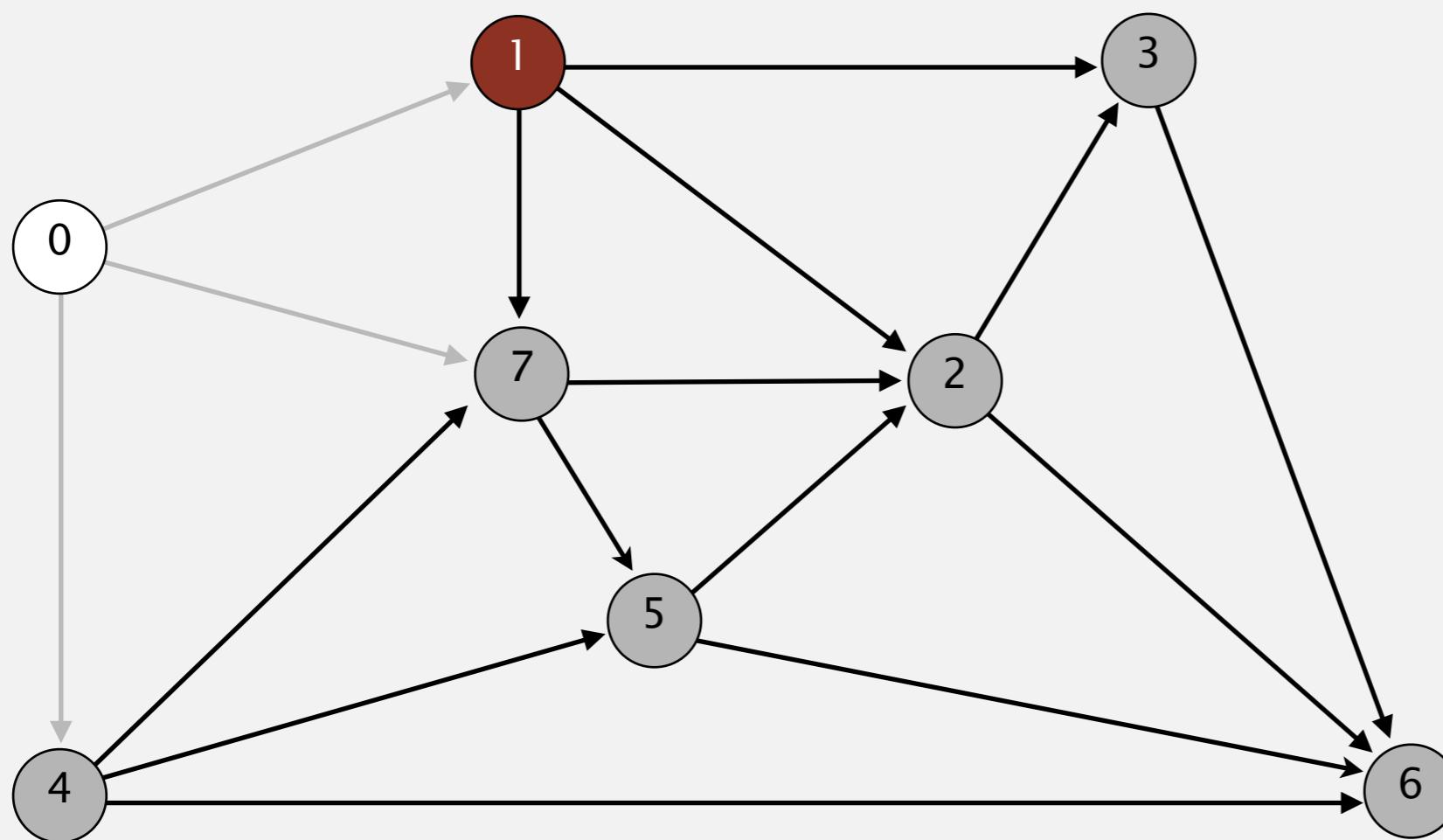
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

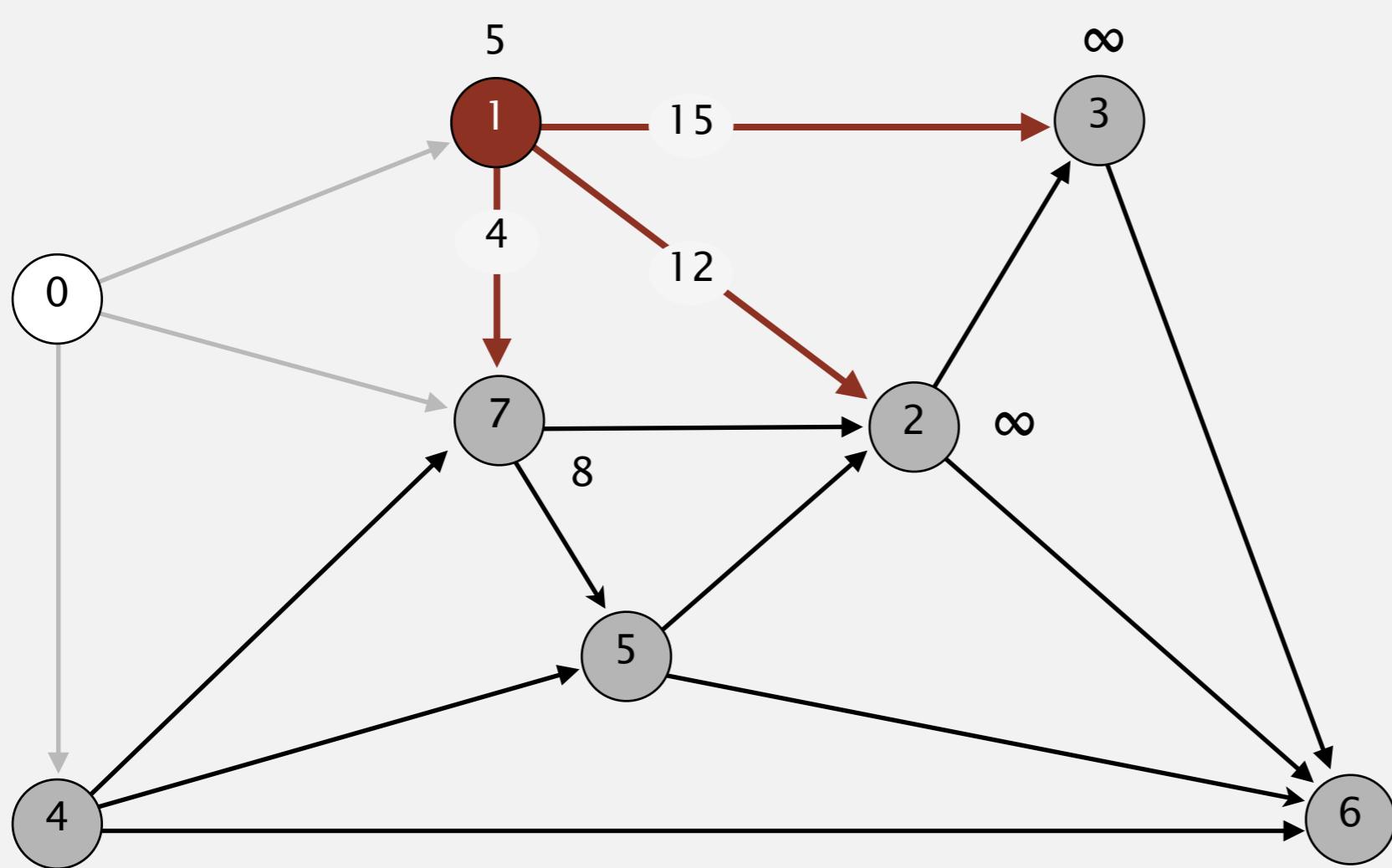


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

choose vertex 1

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

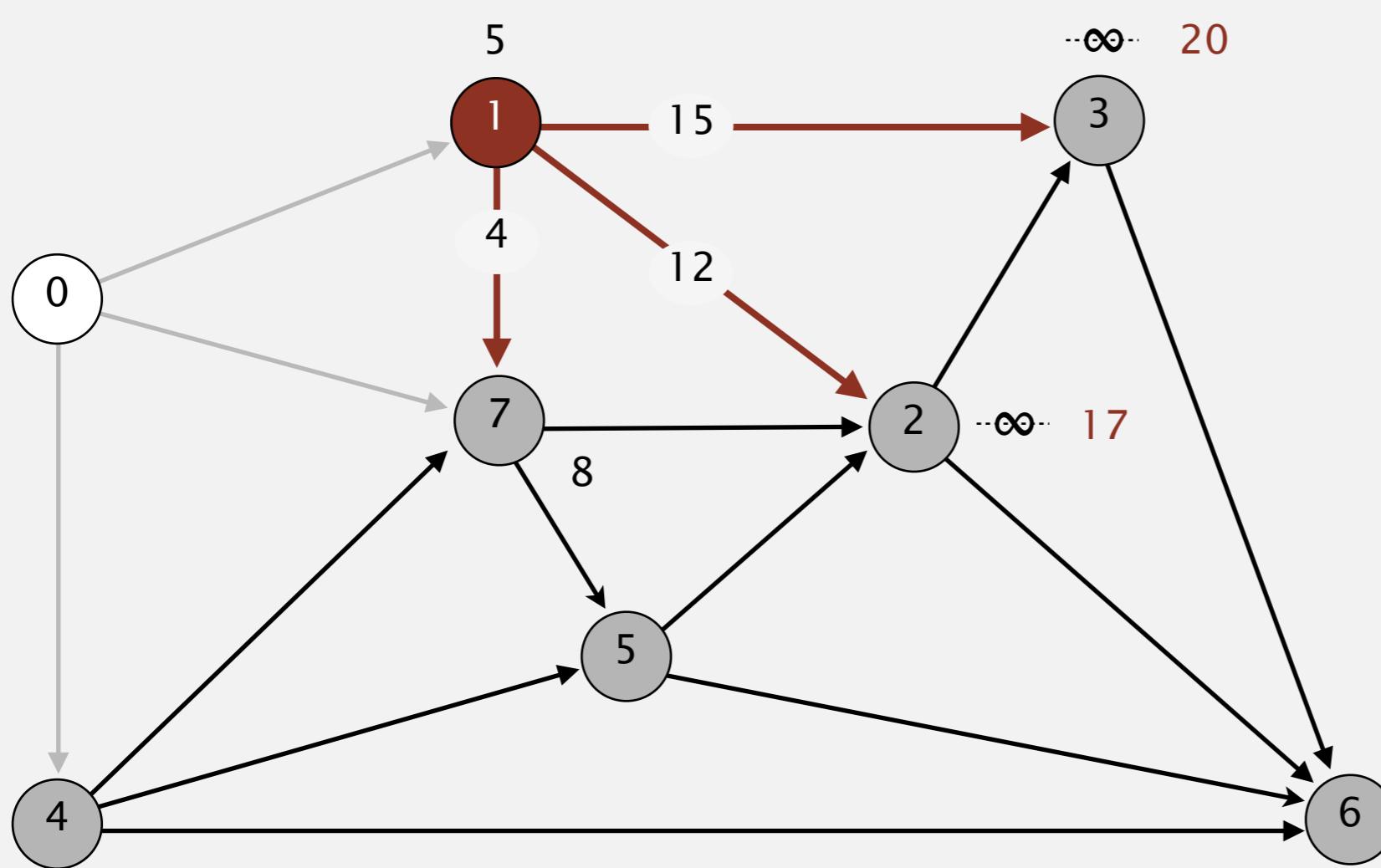


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6	8.0	0→7
7		

relax all edges pointing from 1

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

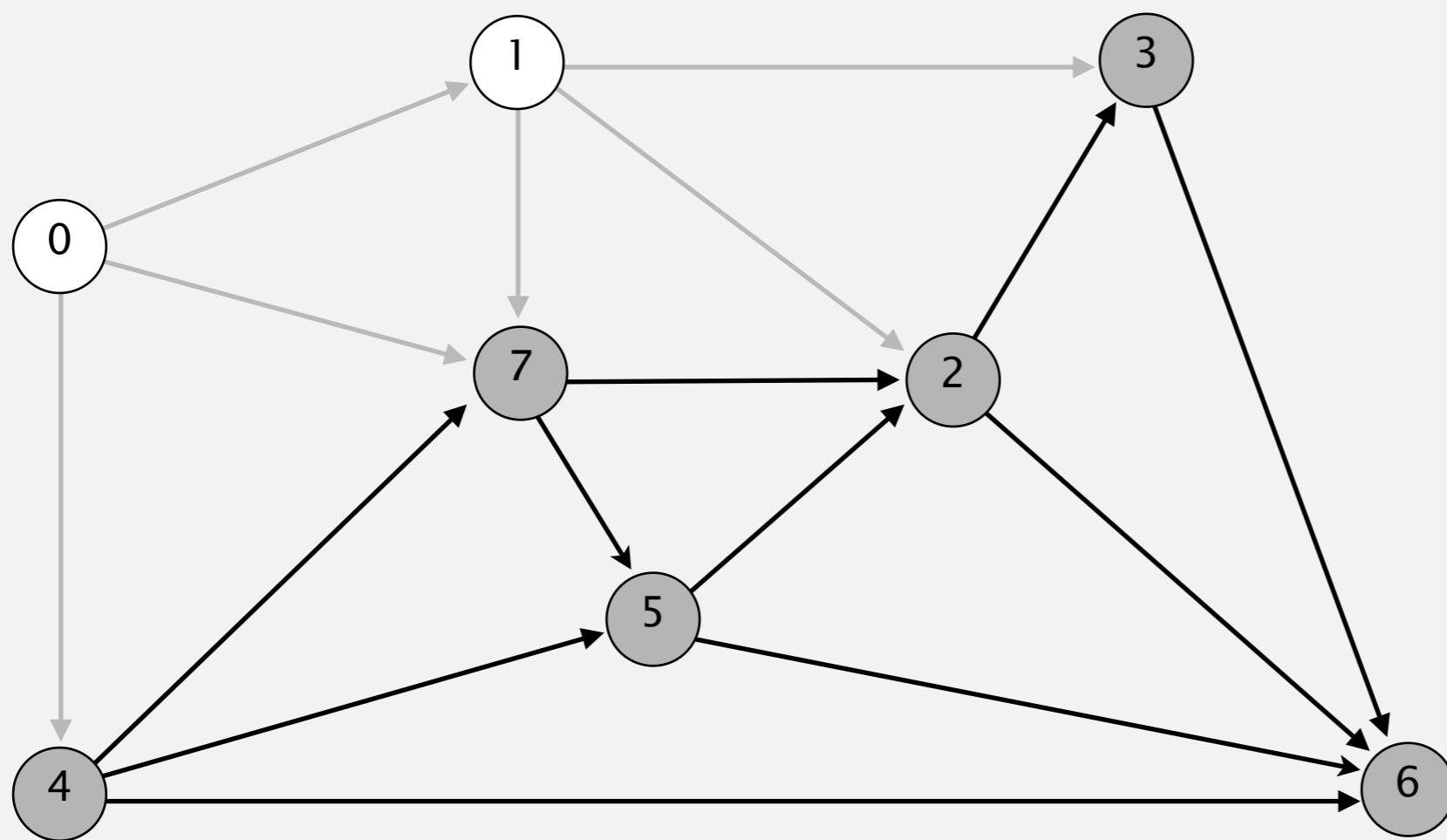


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0 ✓	0→7

relax all edges pointing from 1

Acyclic shortest paths demo

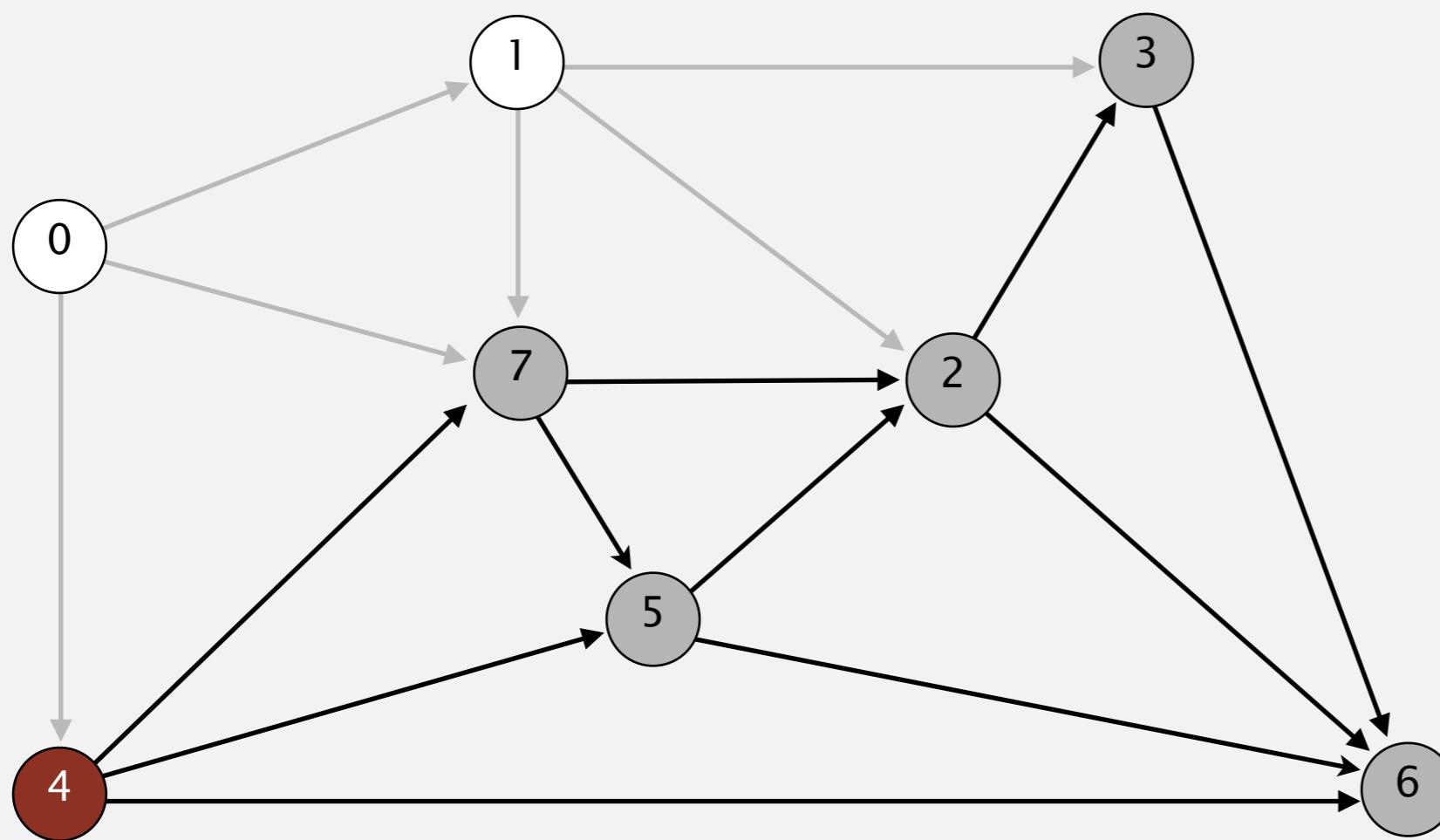
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



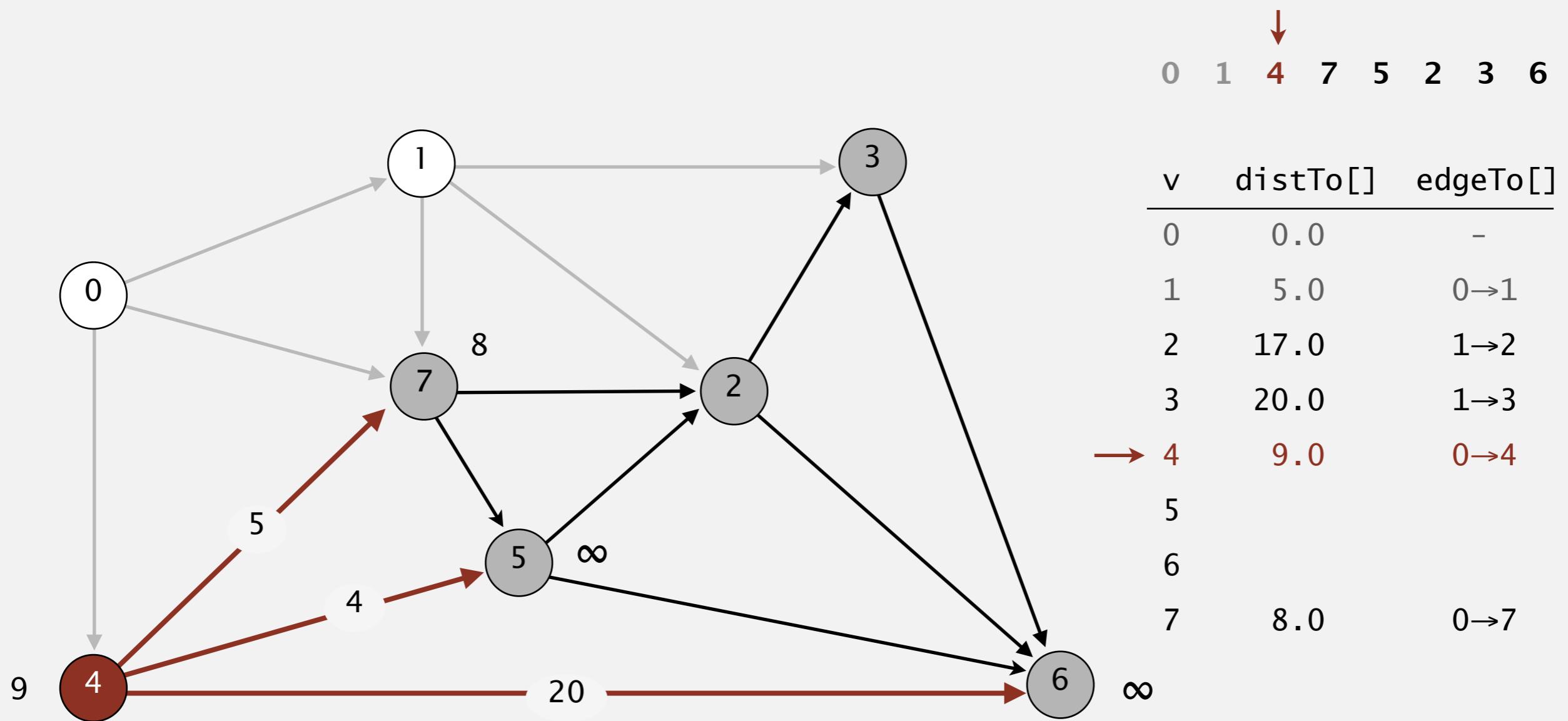
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

select vertex 4

(Dijkstra would have selected vertex 7)

Acyclic shortest paths demo

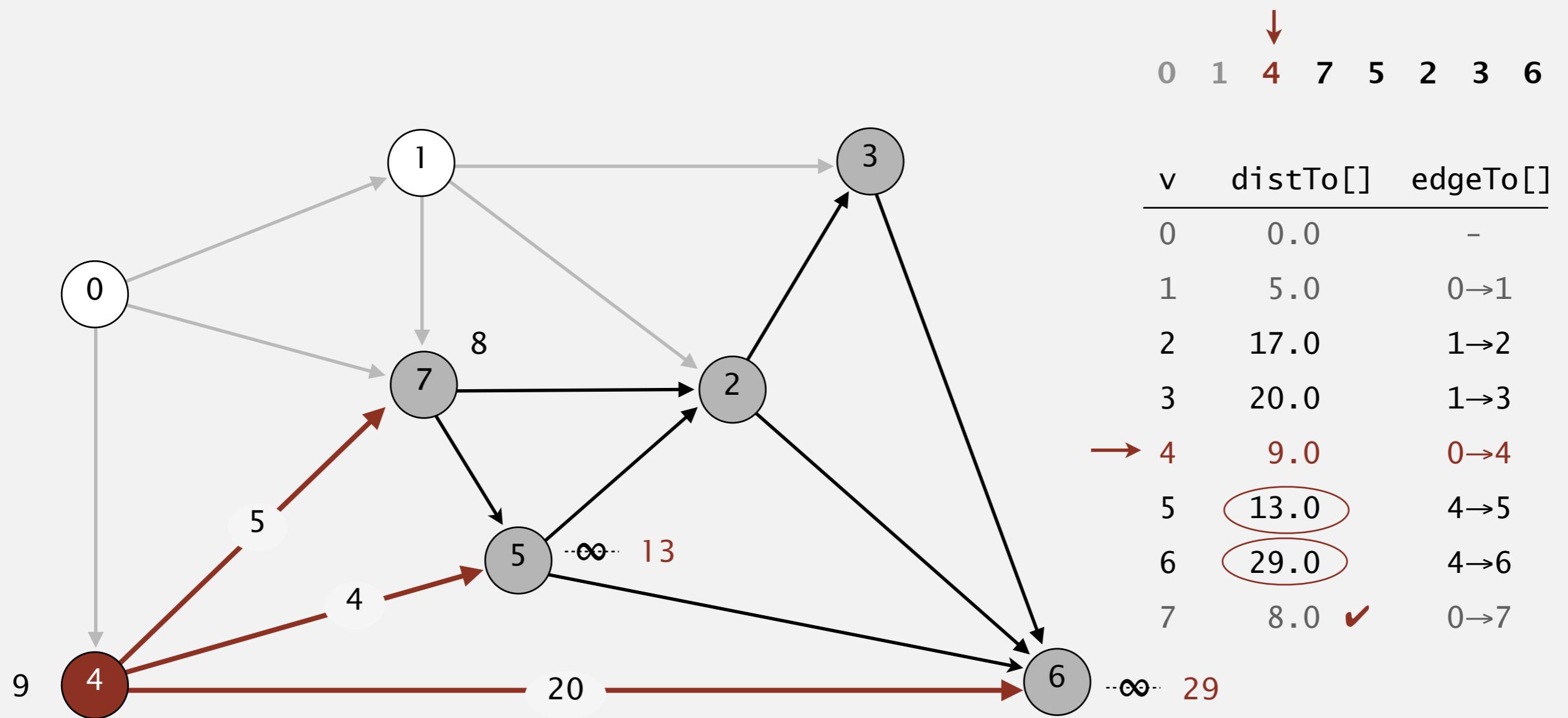
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



relax all edges pointing from 4

Acyclic shortest paths demo

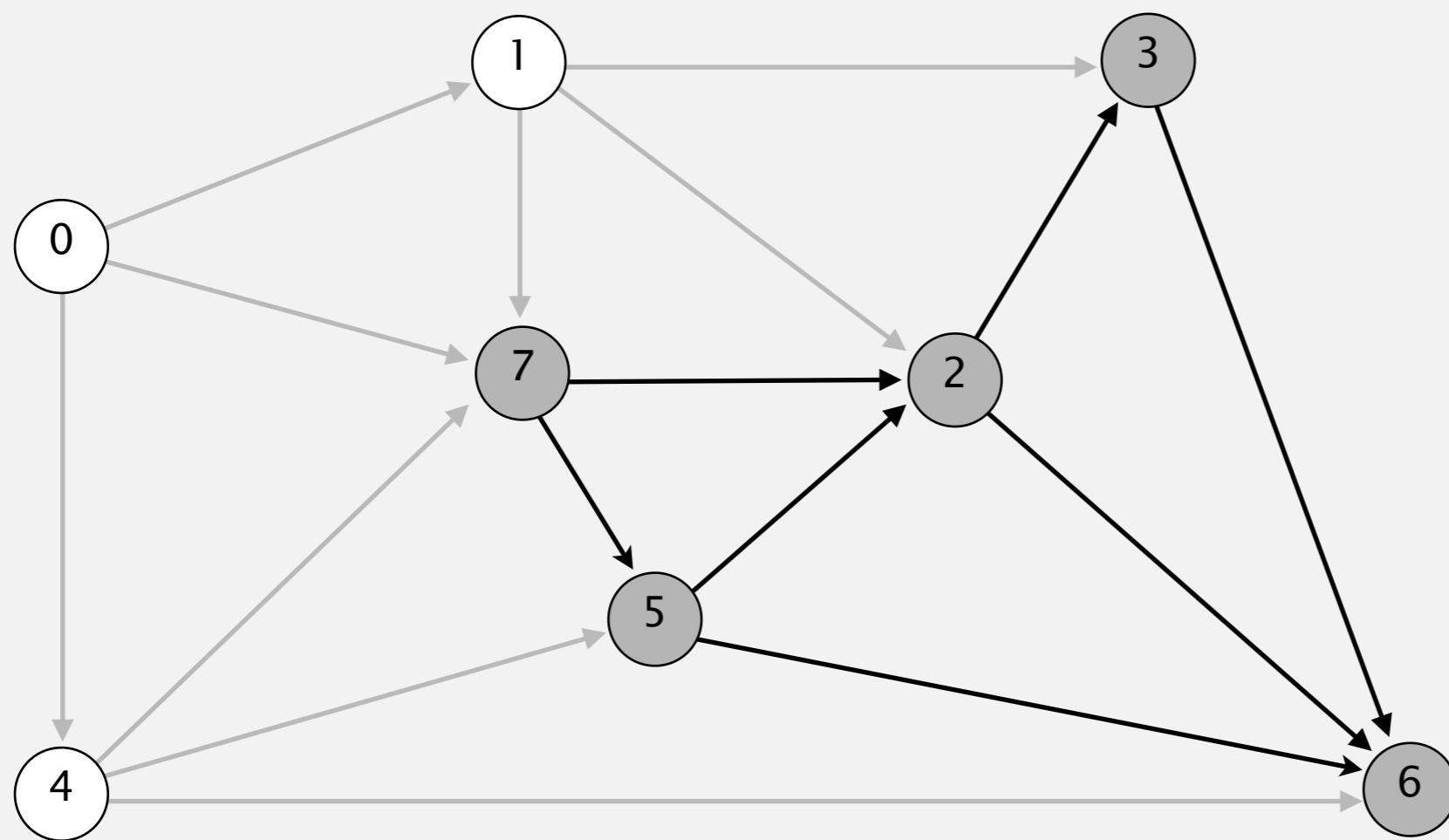
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



relax all edges pointing from 4

Acyclic shortest paths demo

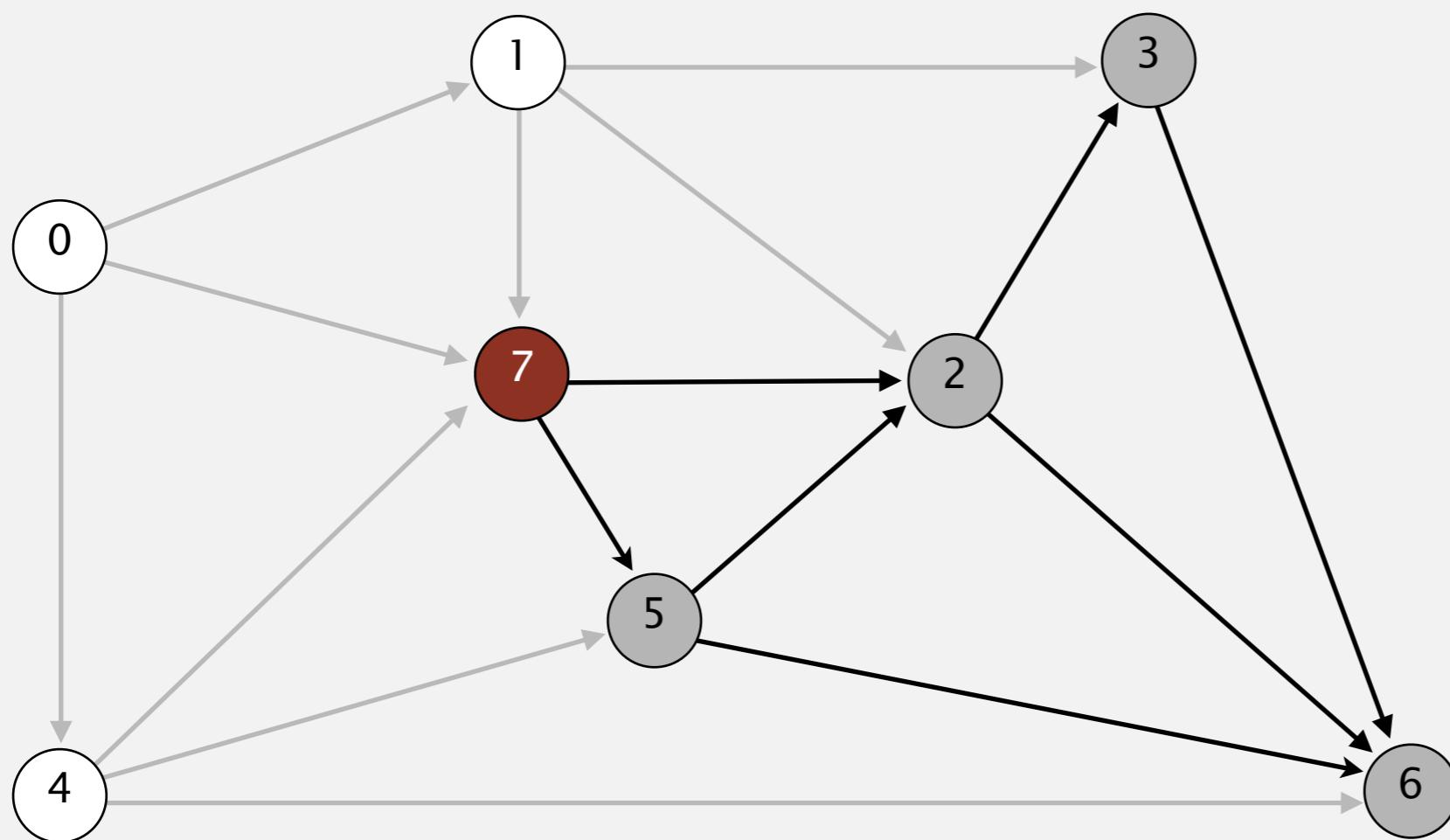
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

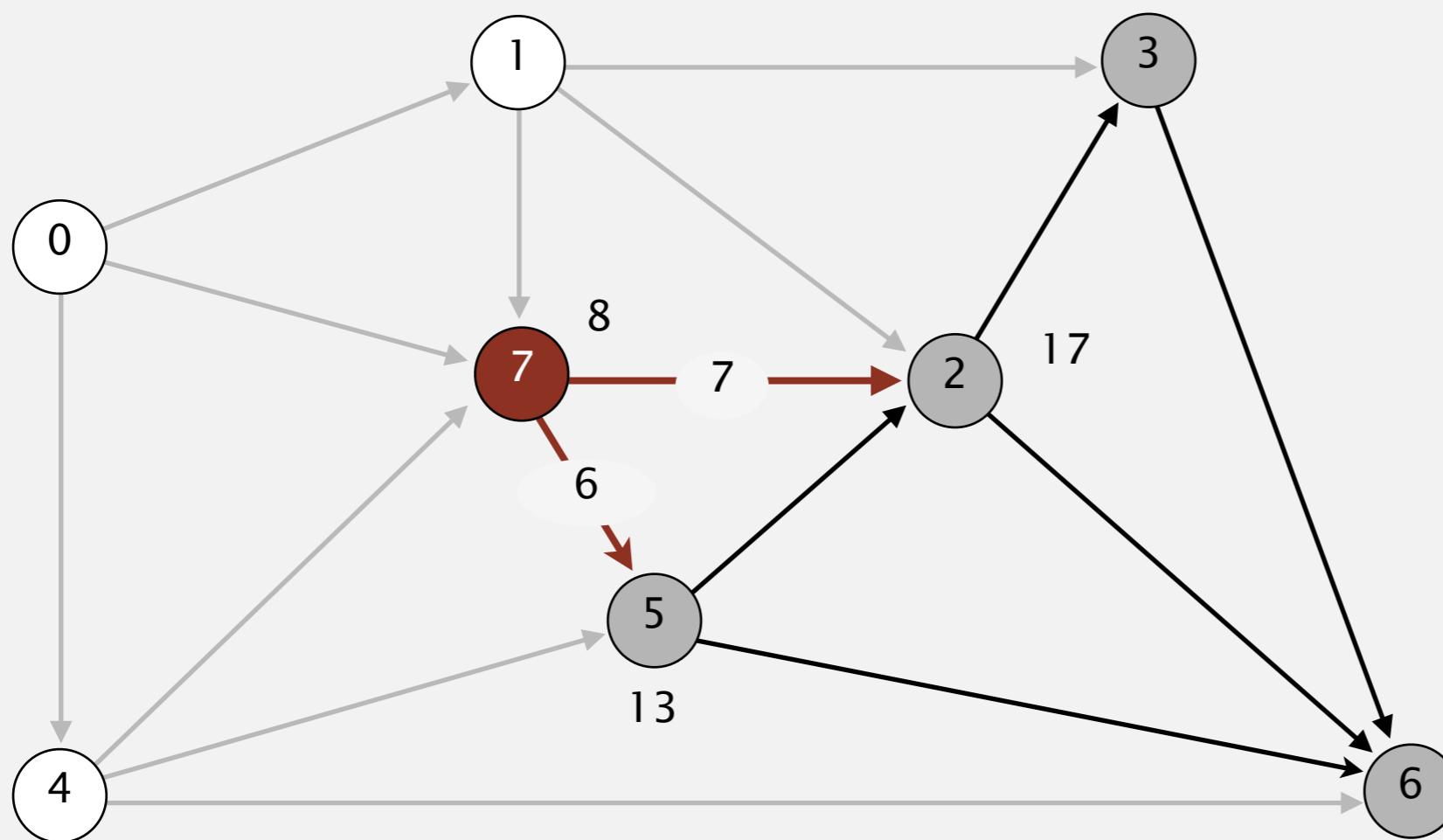


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

choose vertex 7

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

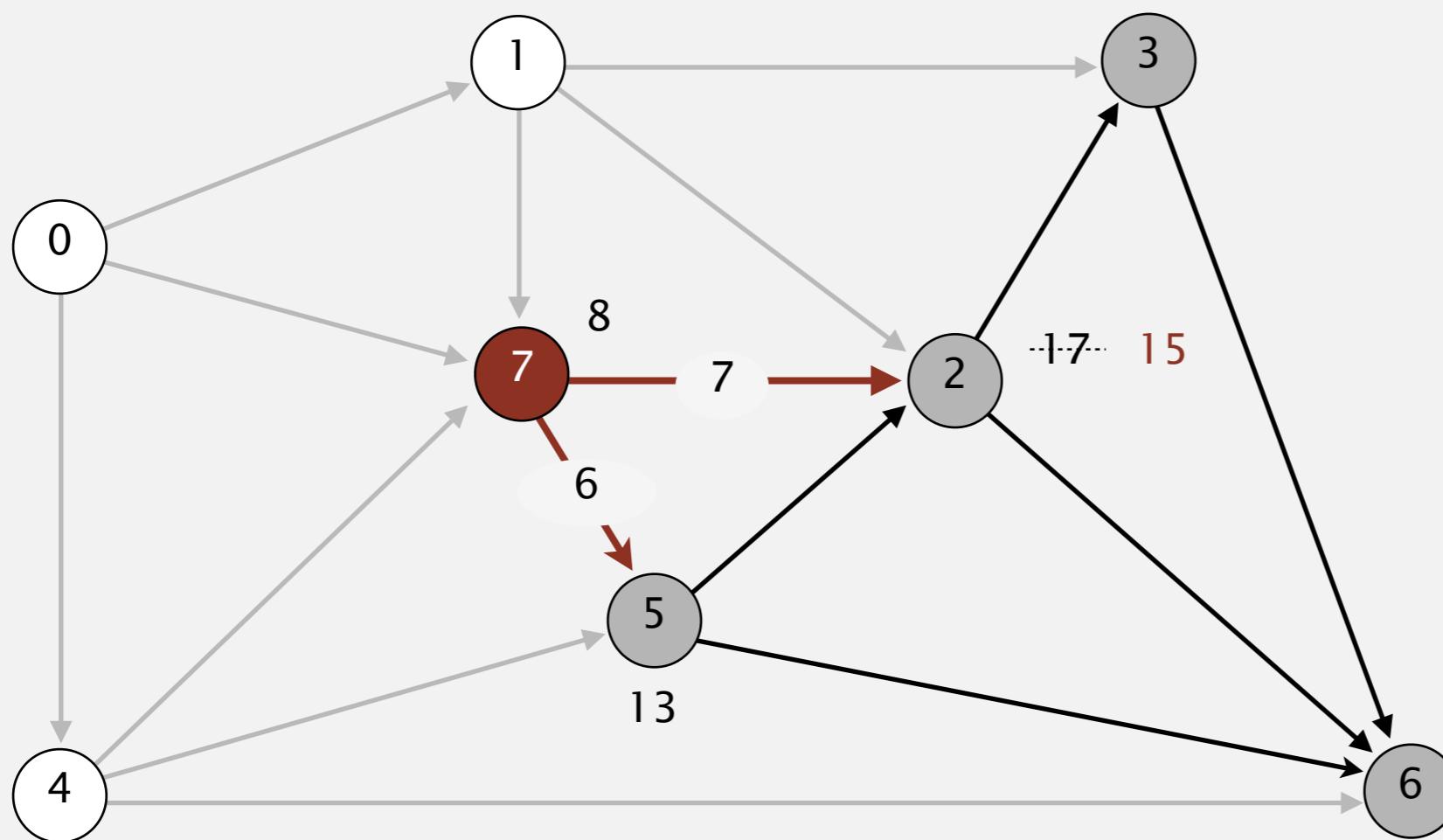


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

relax all edges pointing from 7

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

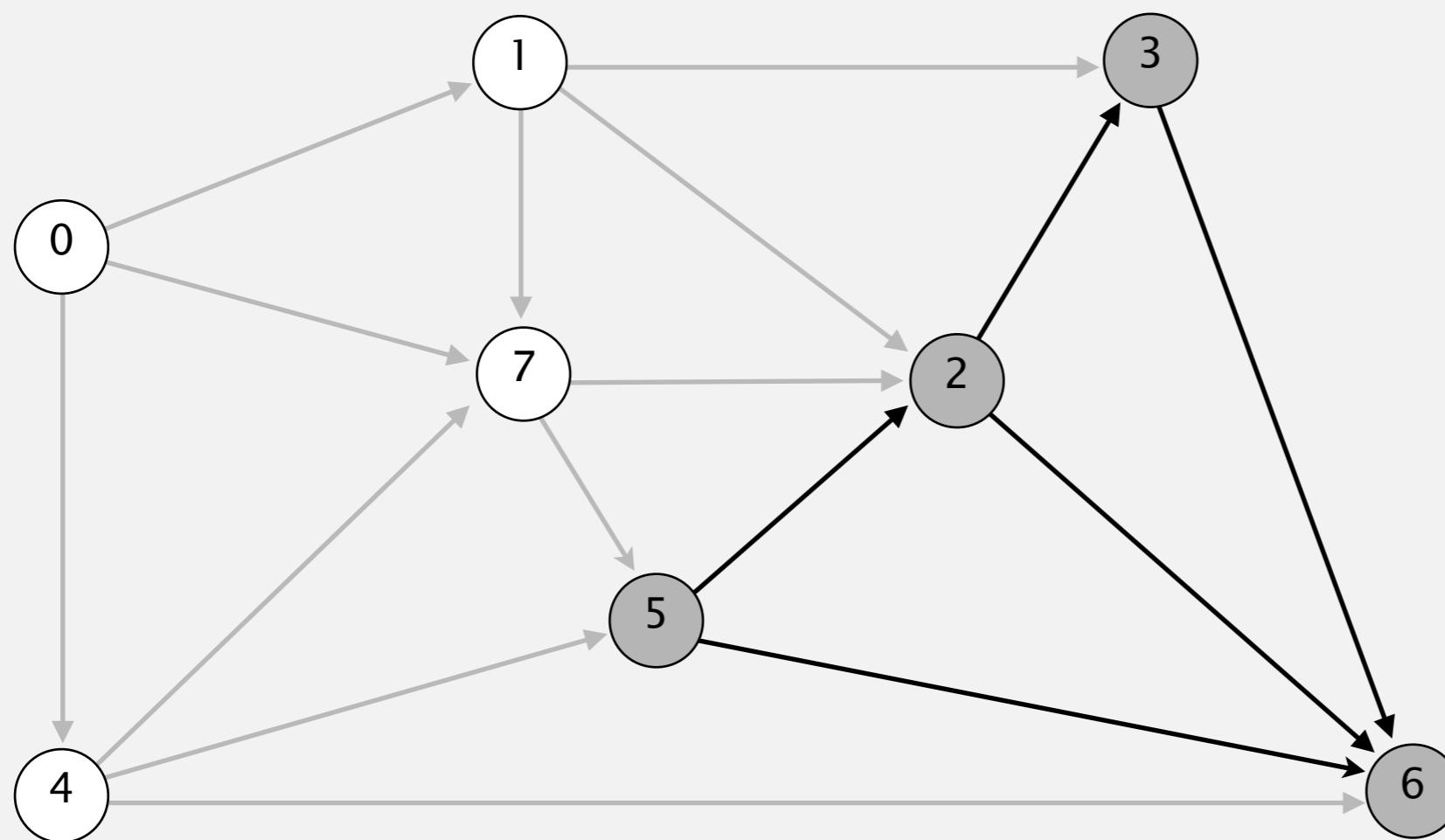


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

relax all edges pointing from 7

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

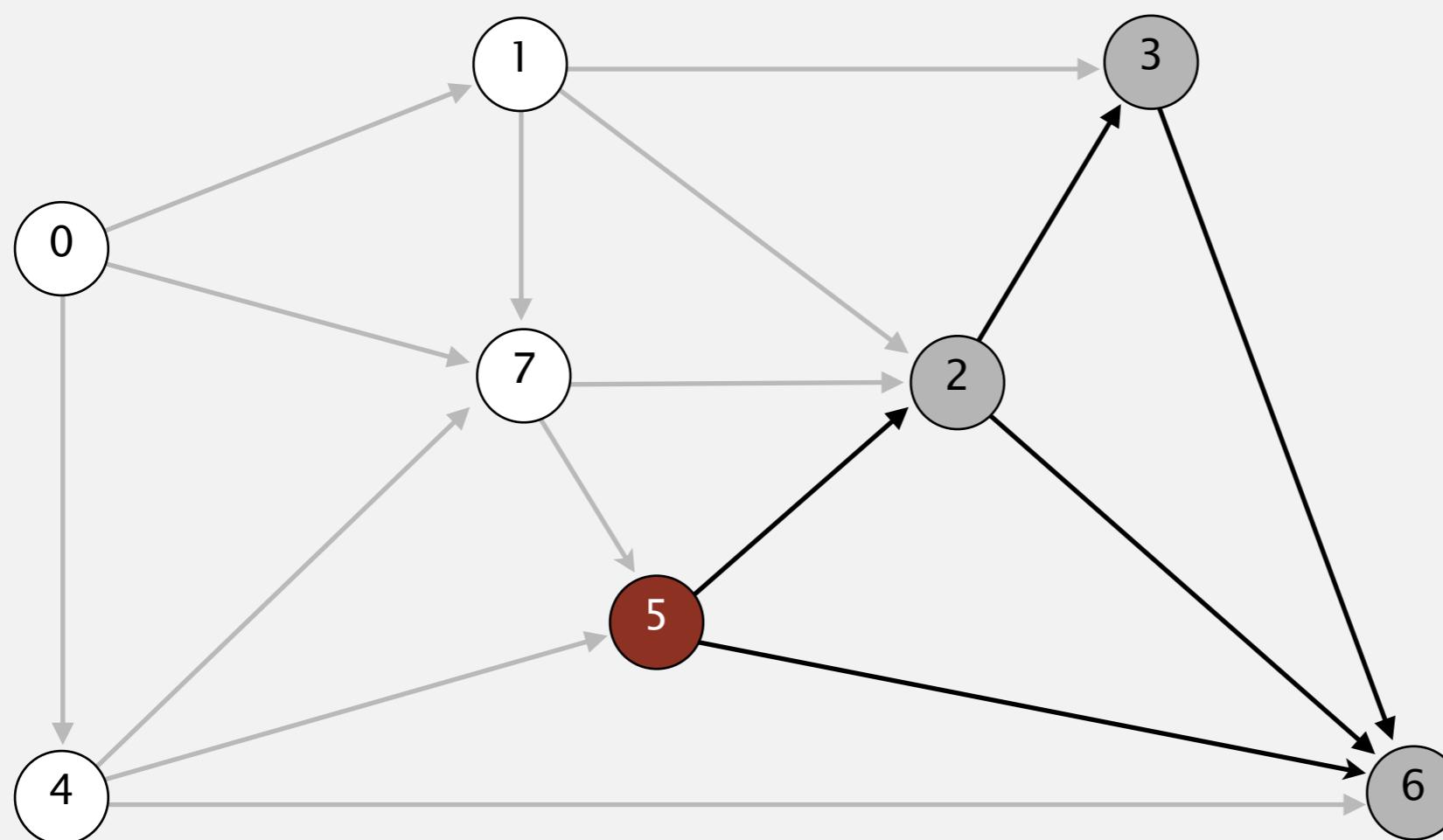


0 1 4 7 5 2 3 6
↓

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

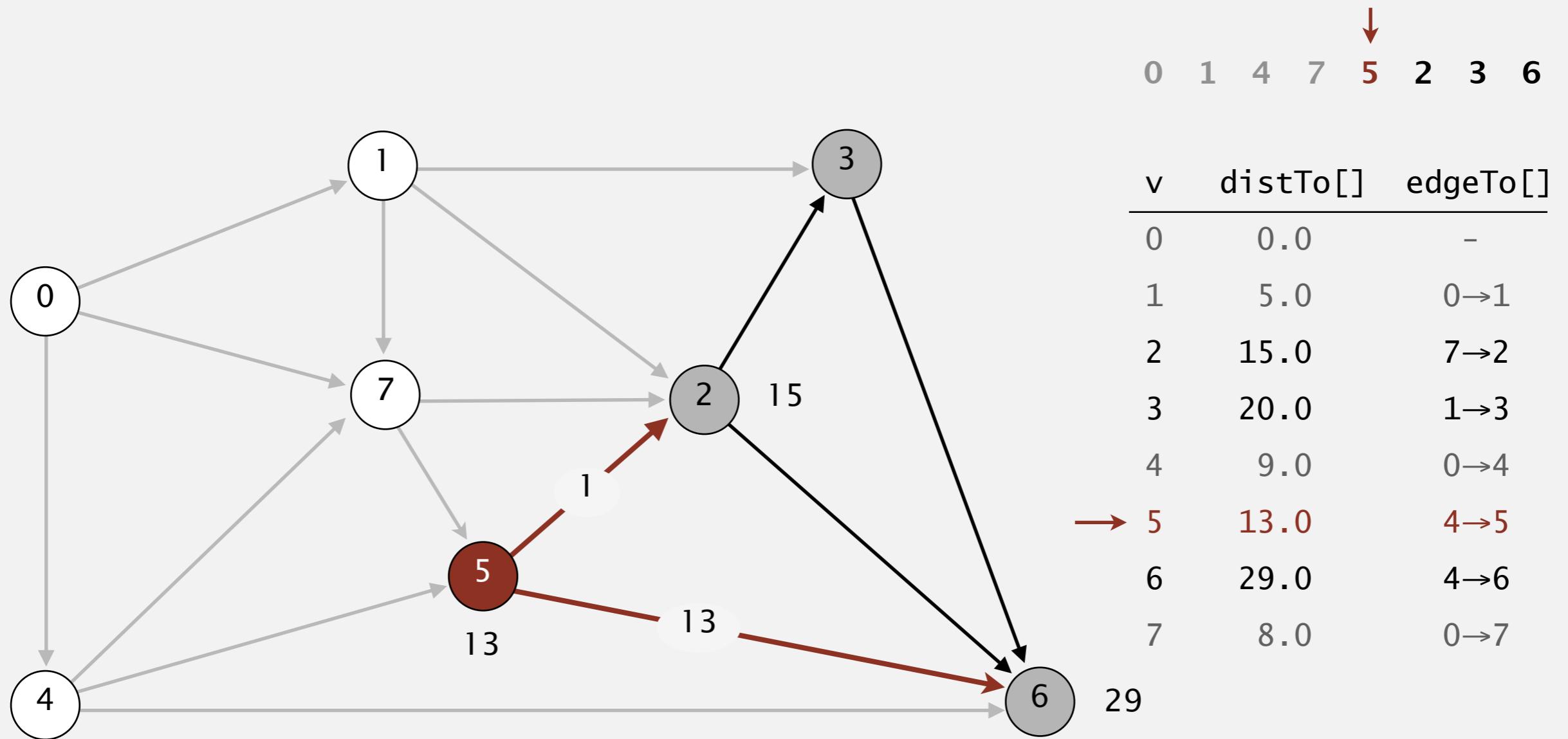


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

select vertex 5

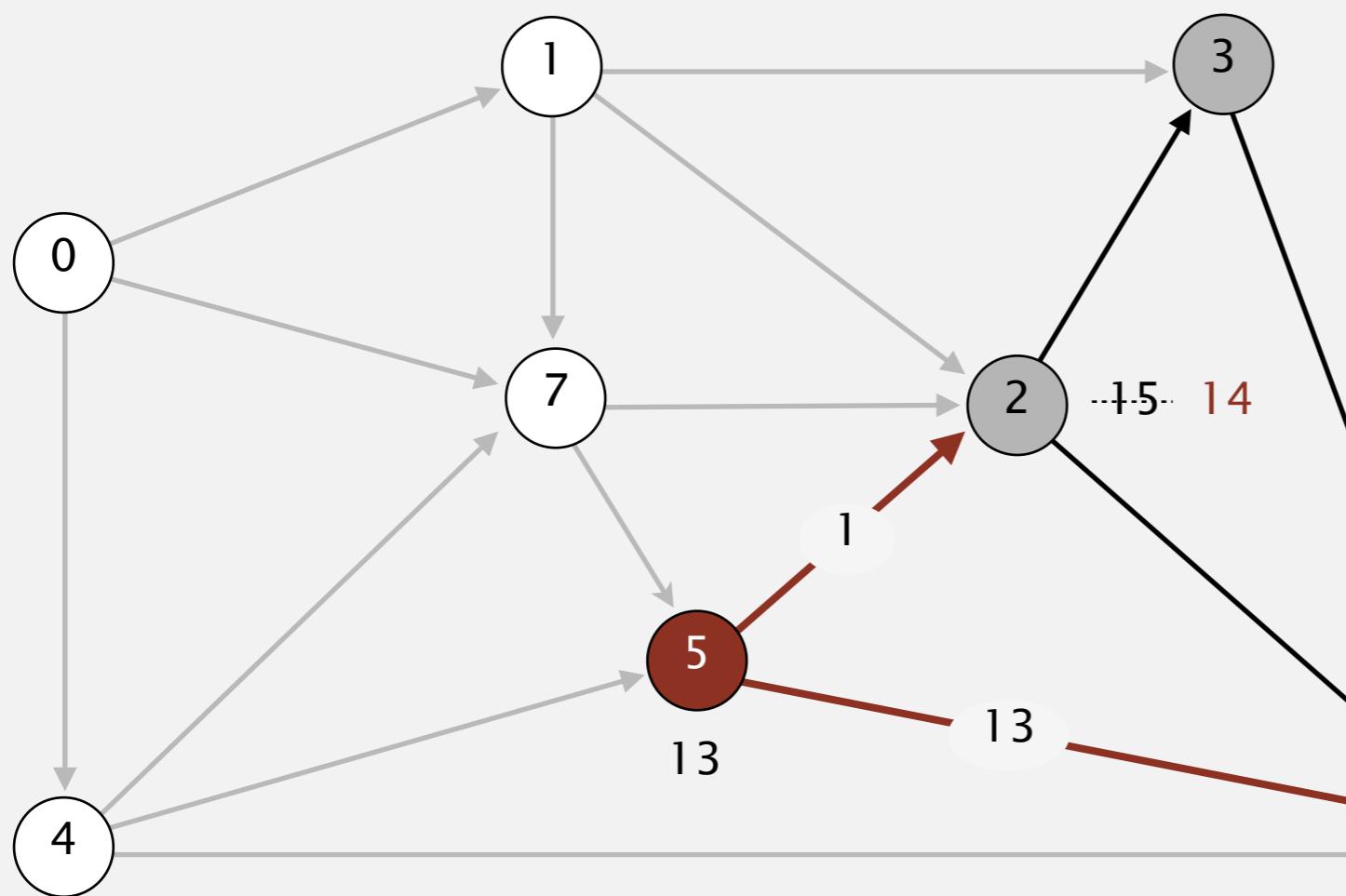
Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

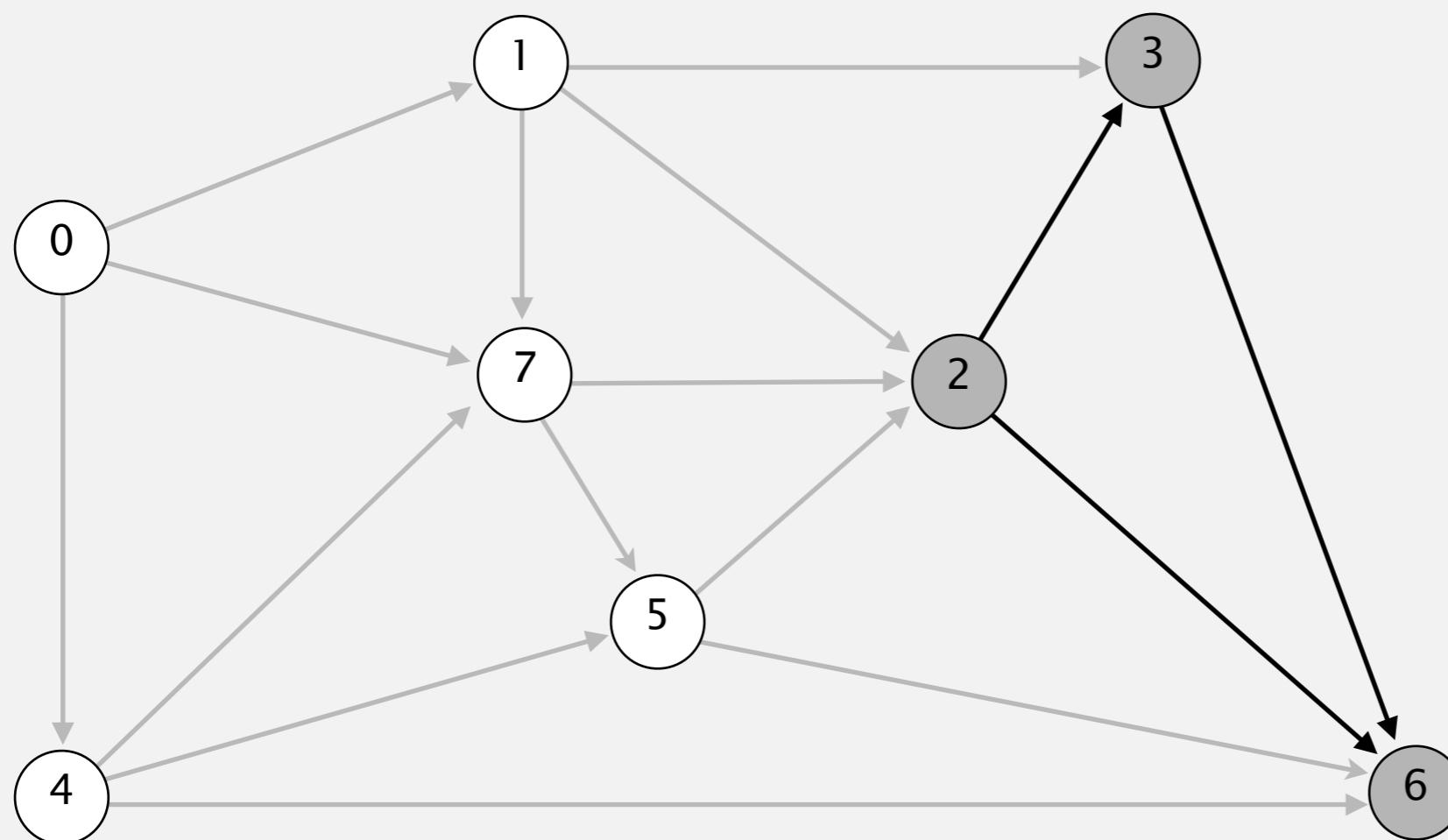


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

relax all edges pointing from 5

Acyclic shortest paths demo

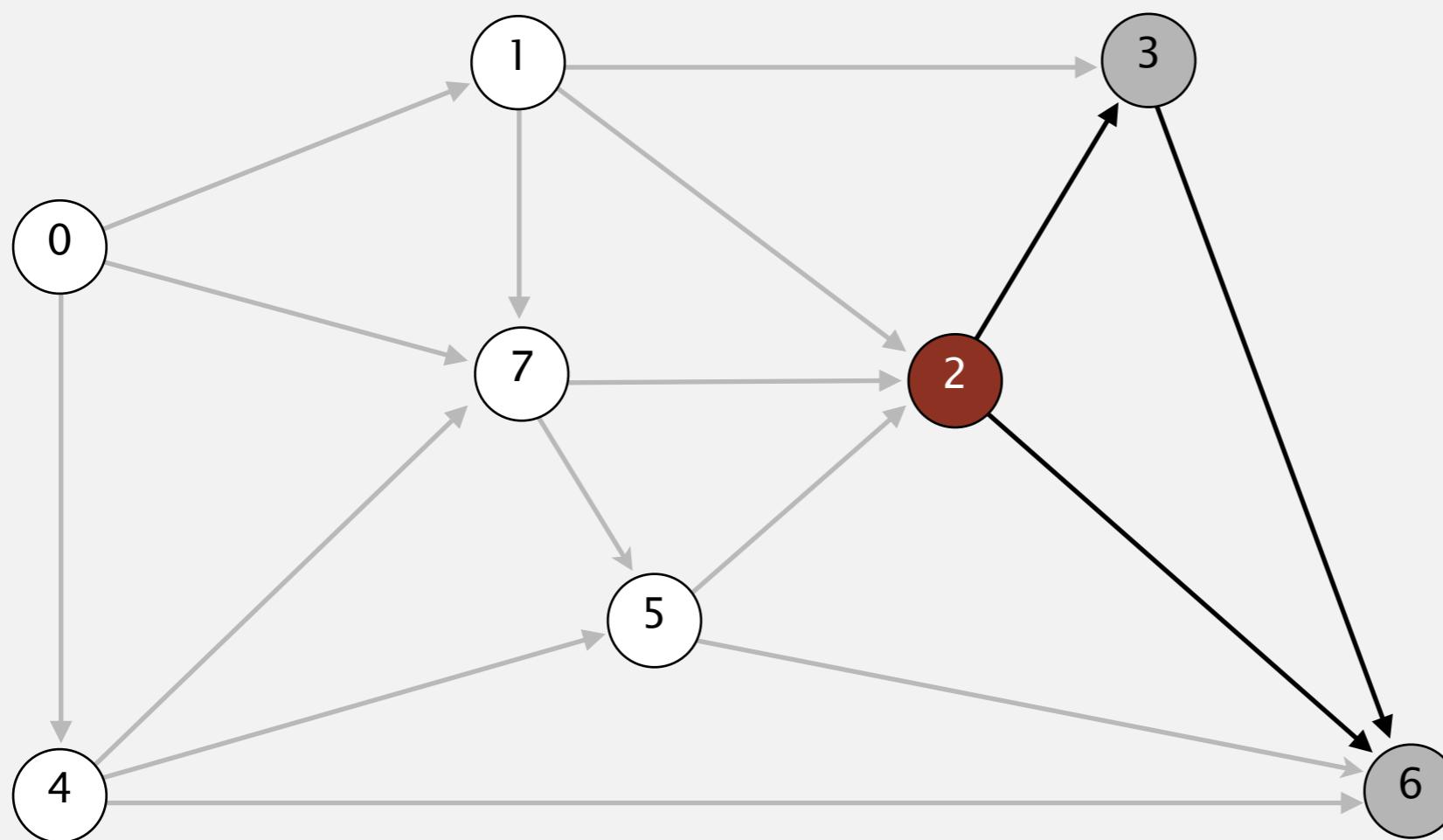
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

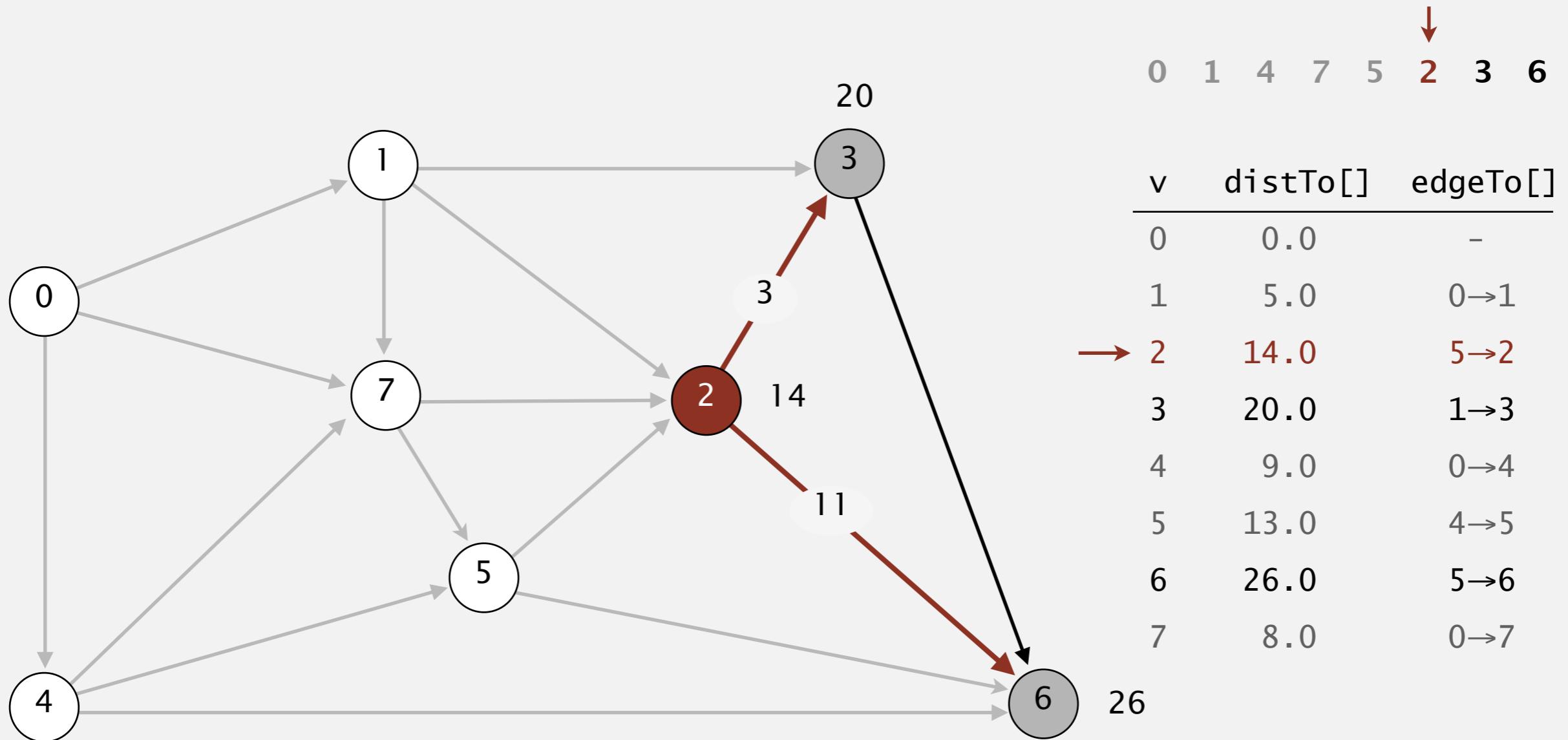


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

select vertex 2

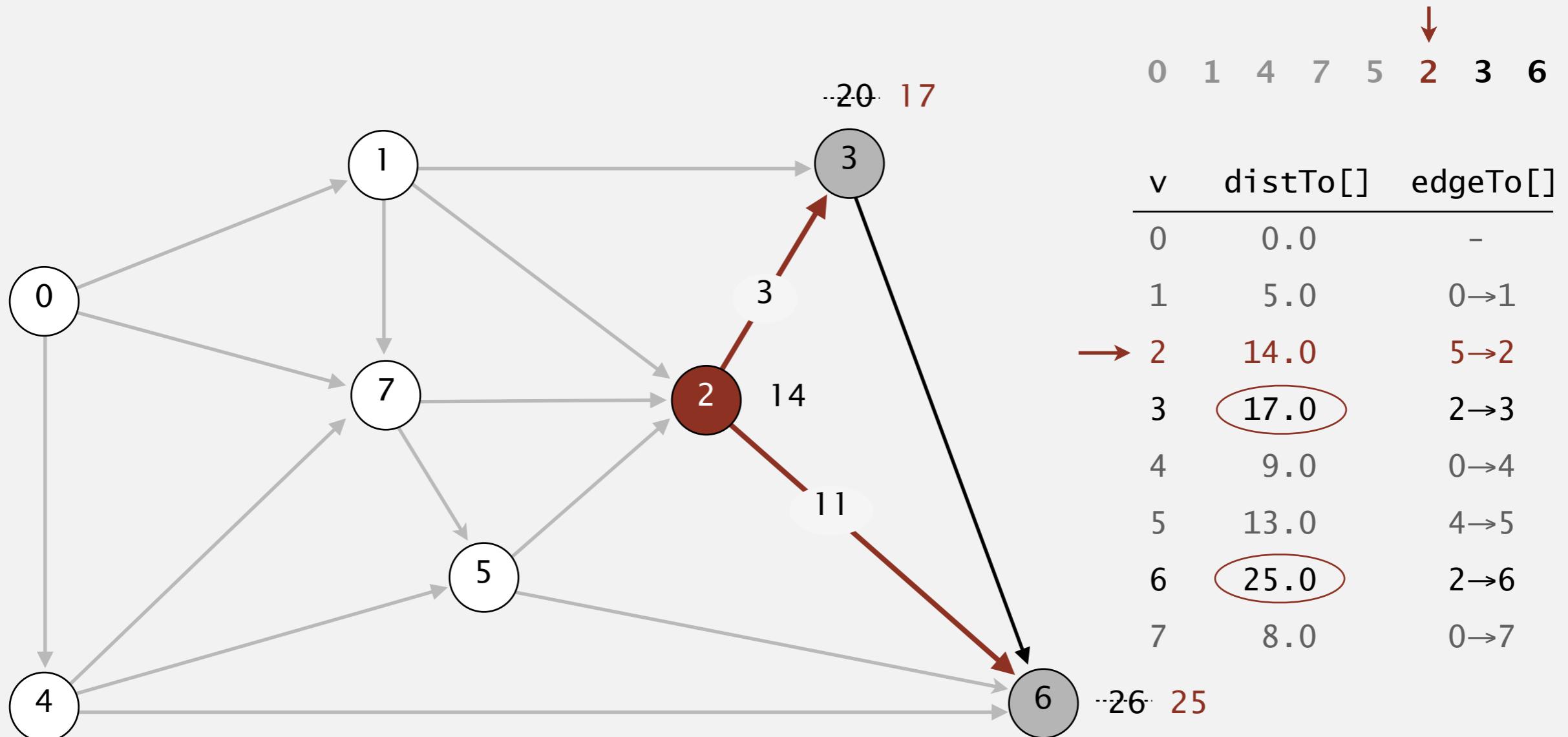
Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



Acyclic shortest paths demo

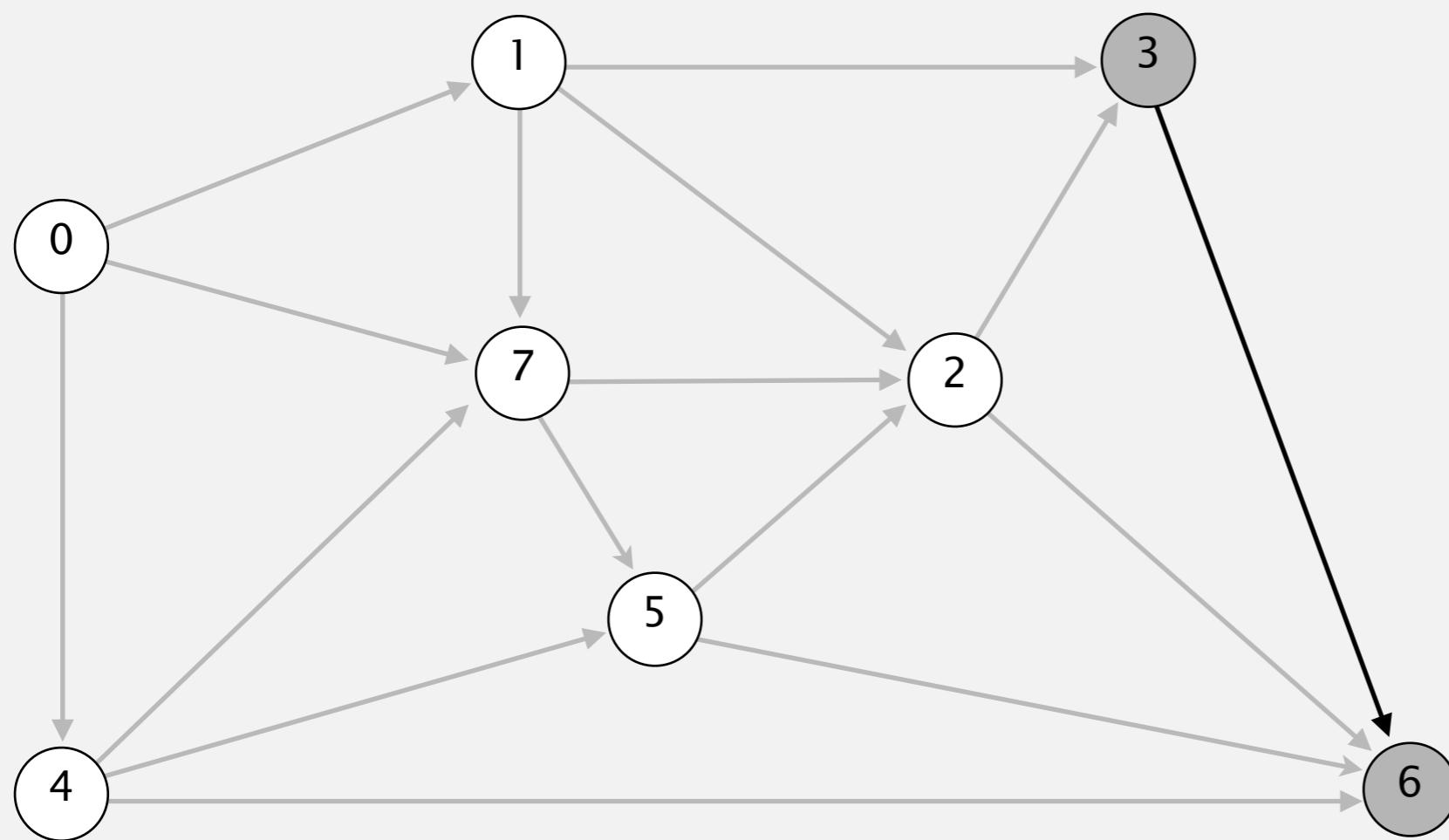
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



relax all edges pointing from 2

Acyclic shortest paths demo

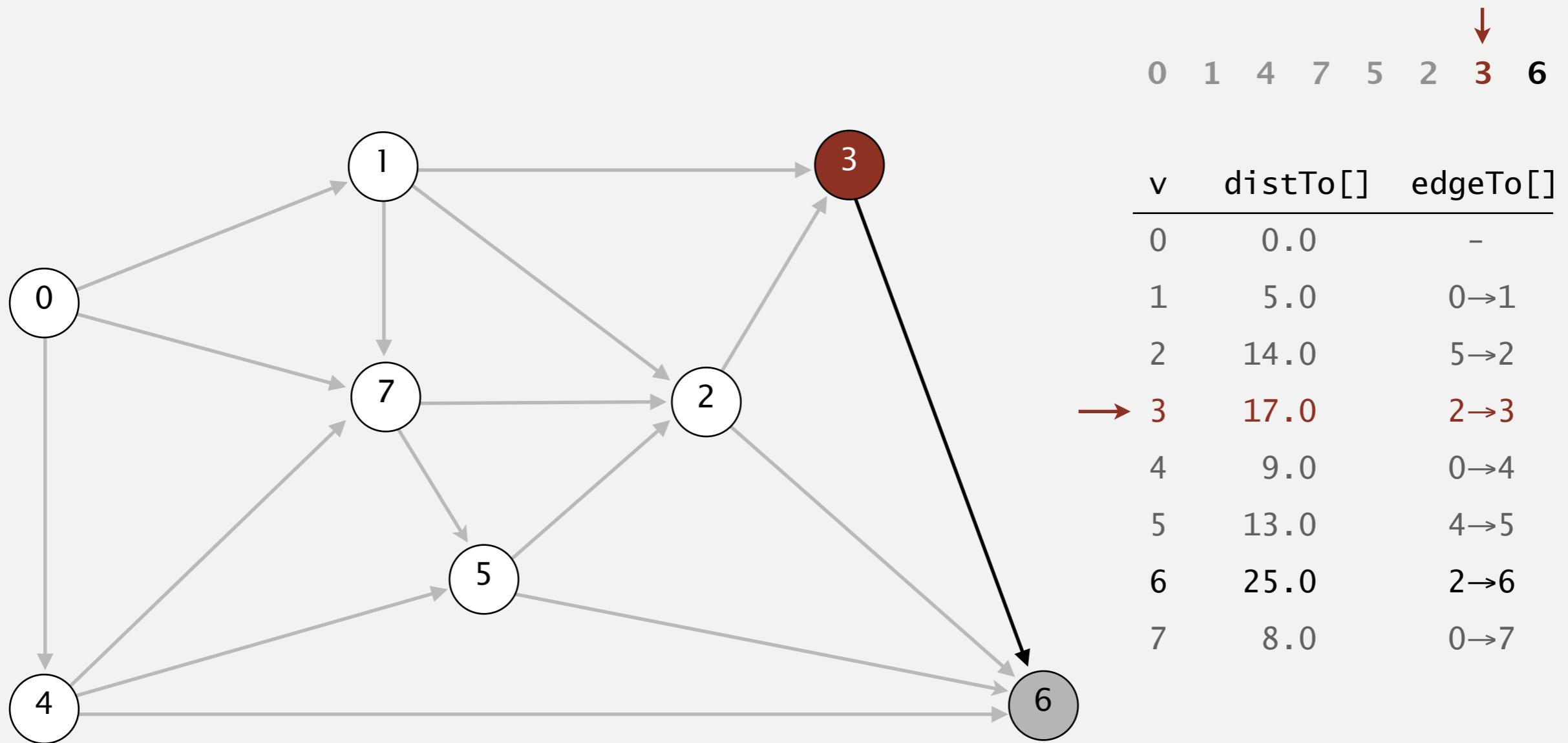
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Acyclic shortest paths demo

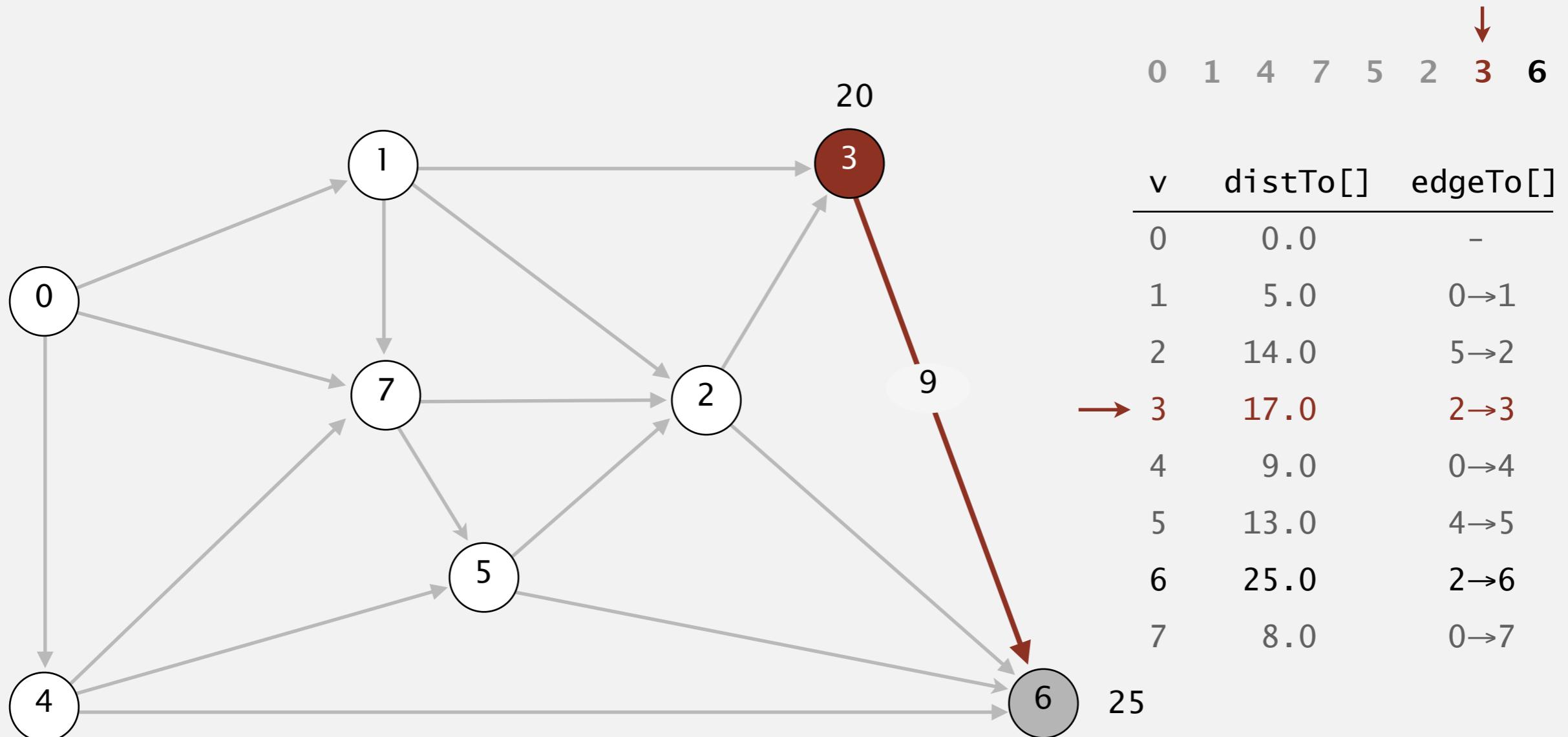
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



select vertex 3

Acyclic shortest paths demo

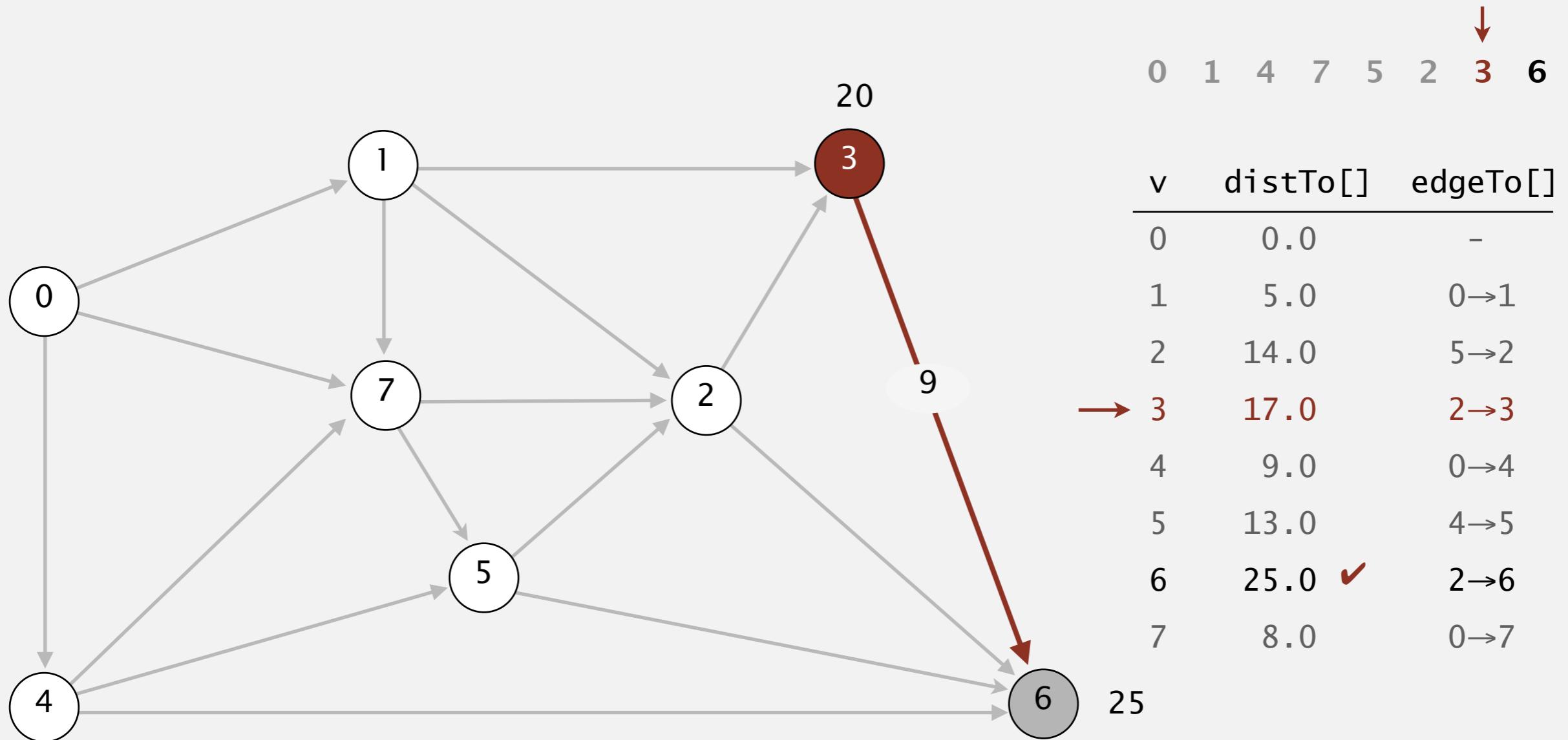
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



relax all edges pointing from 3

Acyclic shortest paths demo

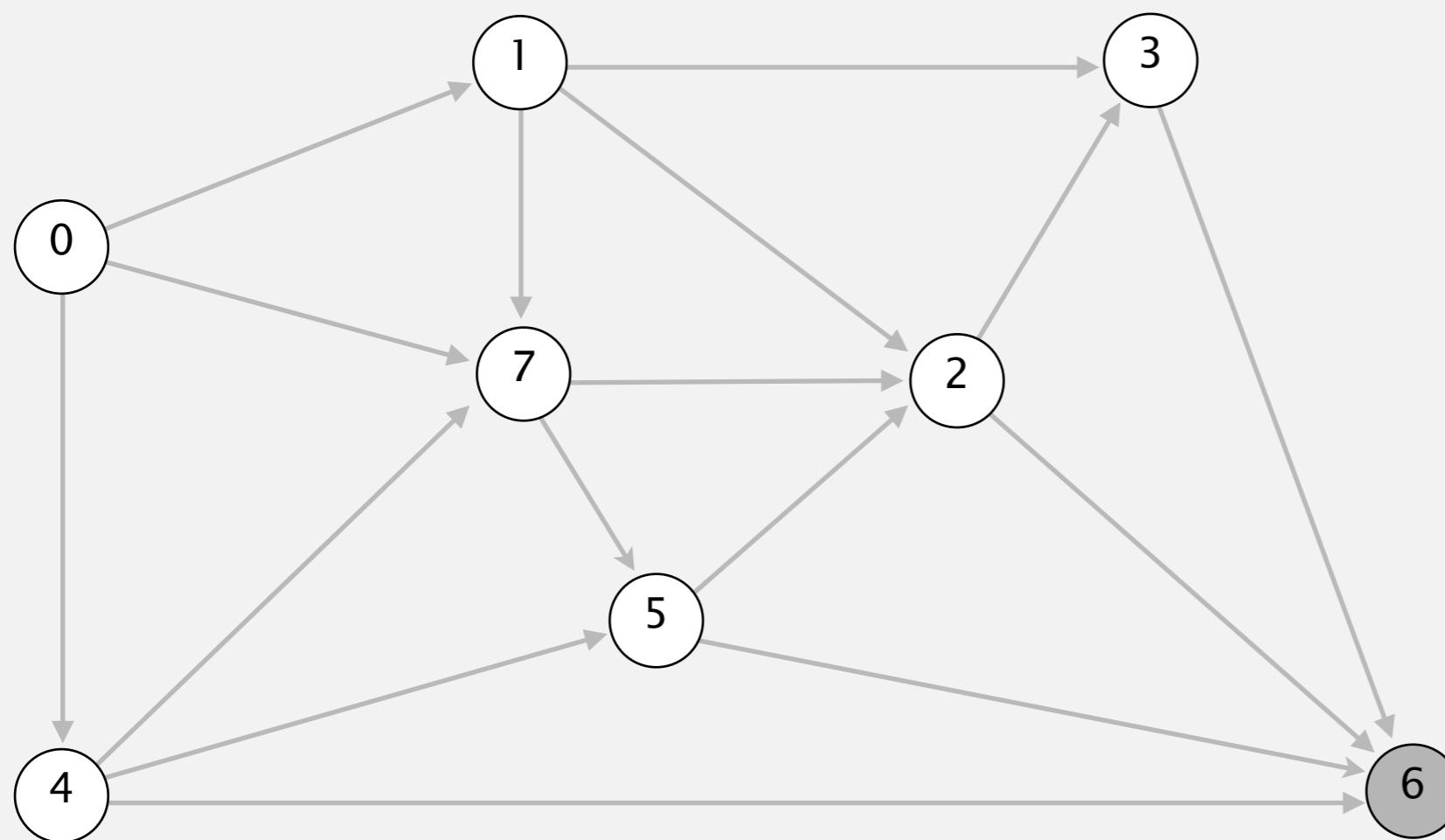
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



relax all edges pointing from 3

Acyclic shortest paths demo

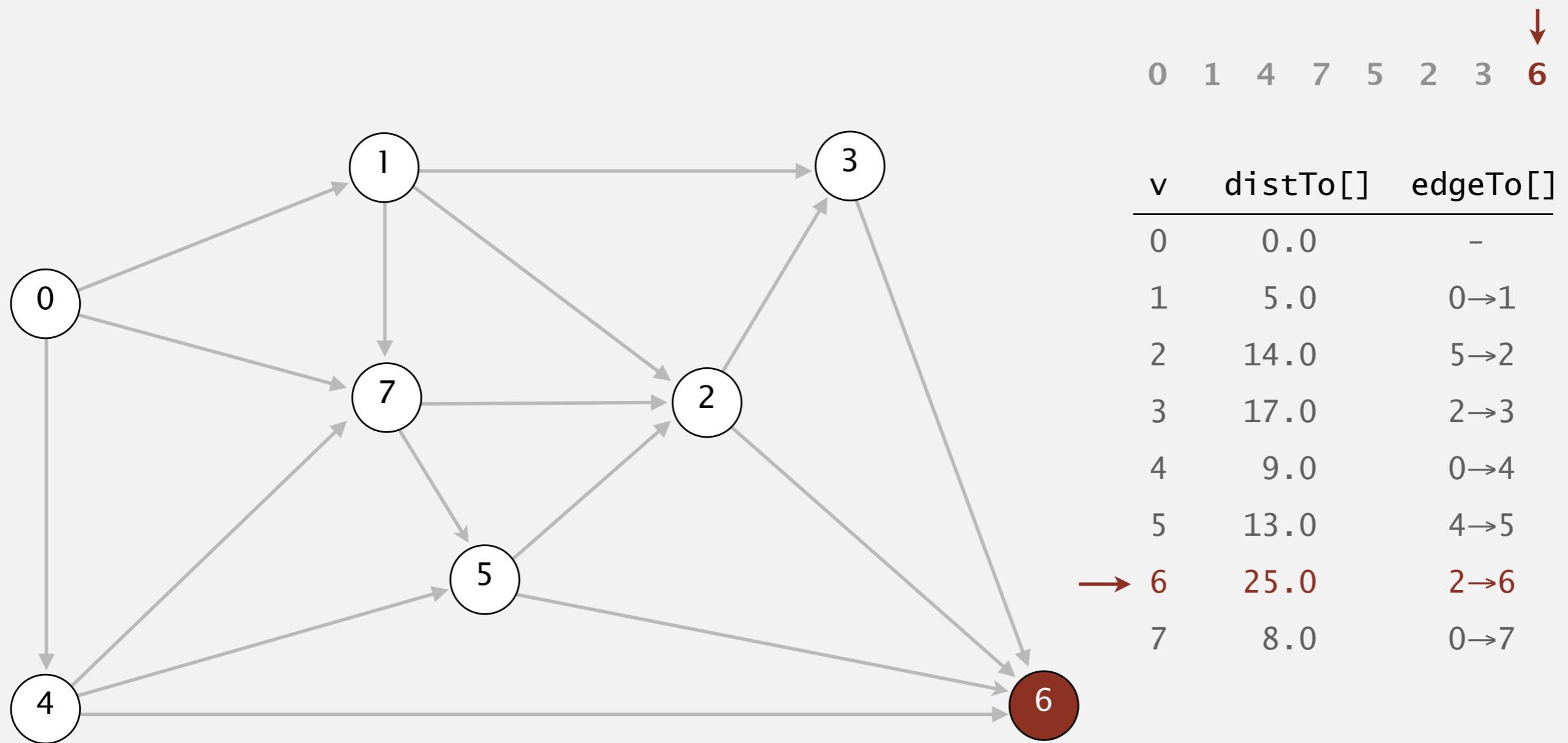
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Acyclic shortest paths demo

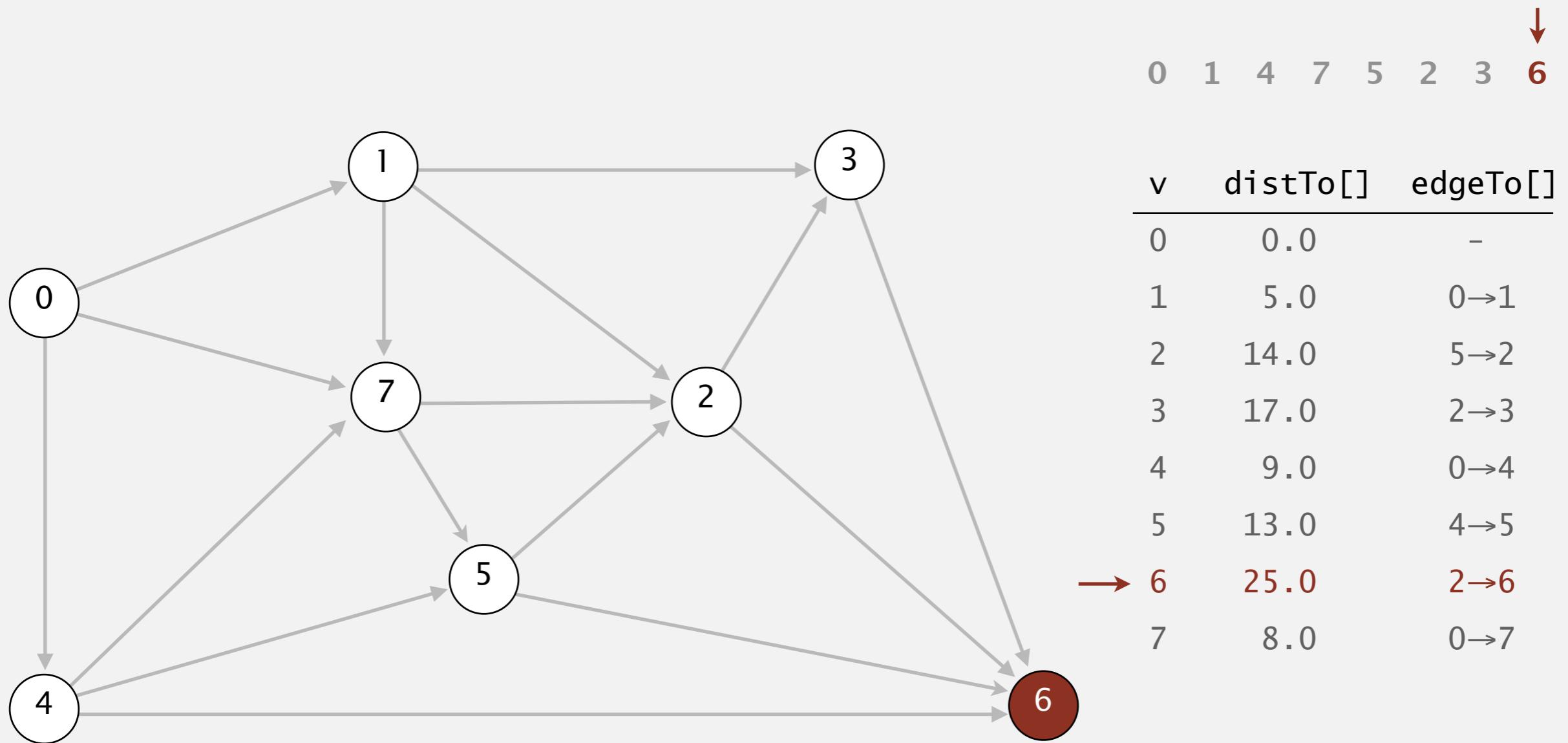
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



select vertex 6

Acyclic shortest paths demo

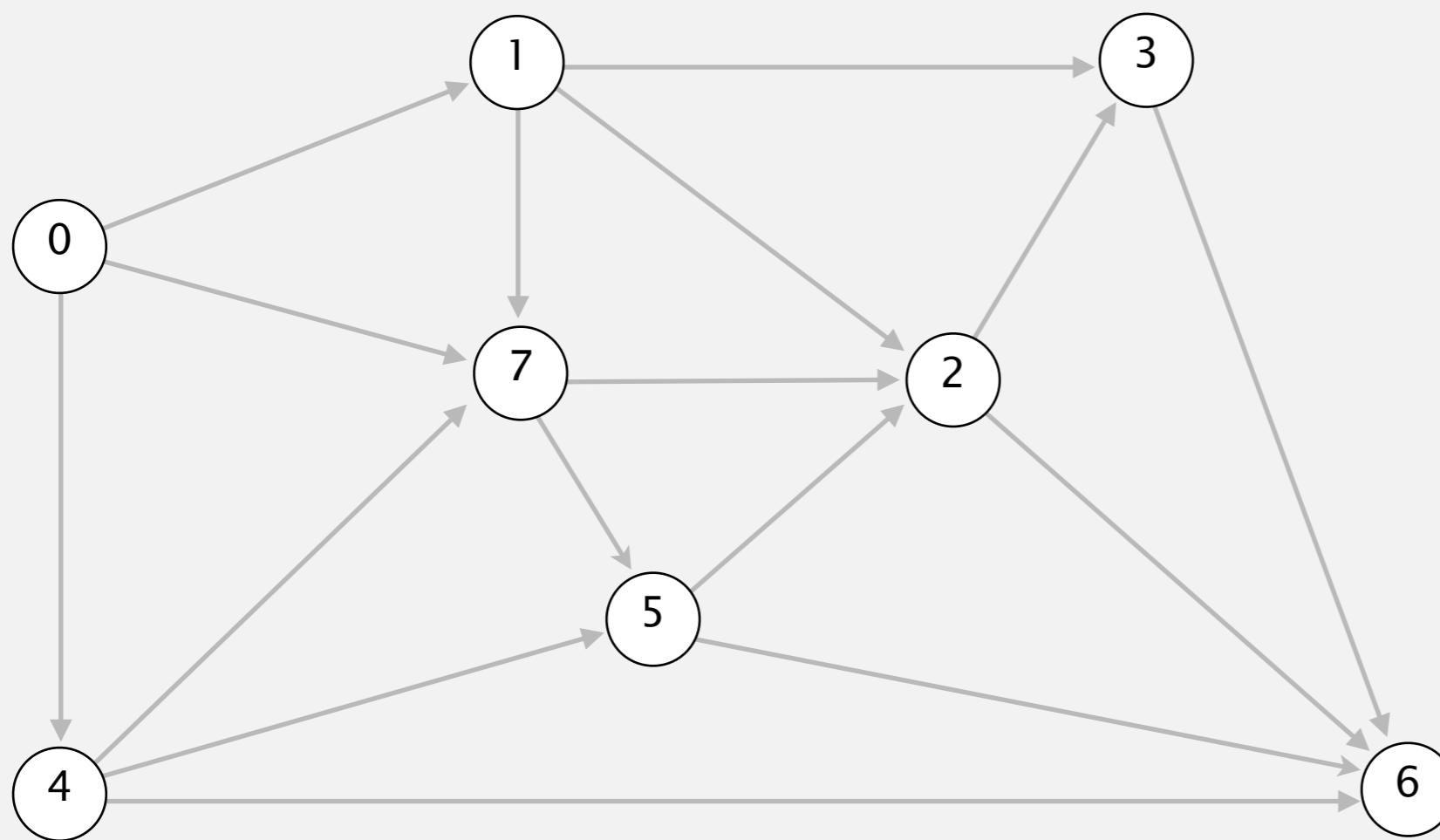
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



relax all edges pointing from 6

Acyclic shortest paths demo

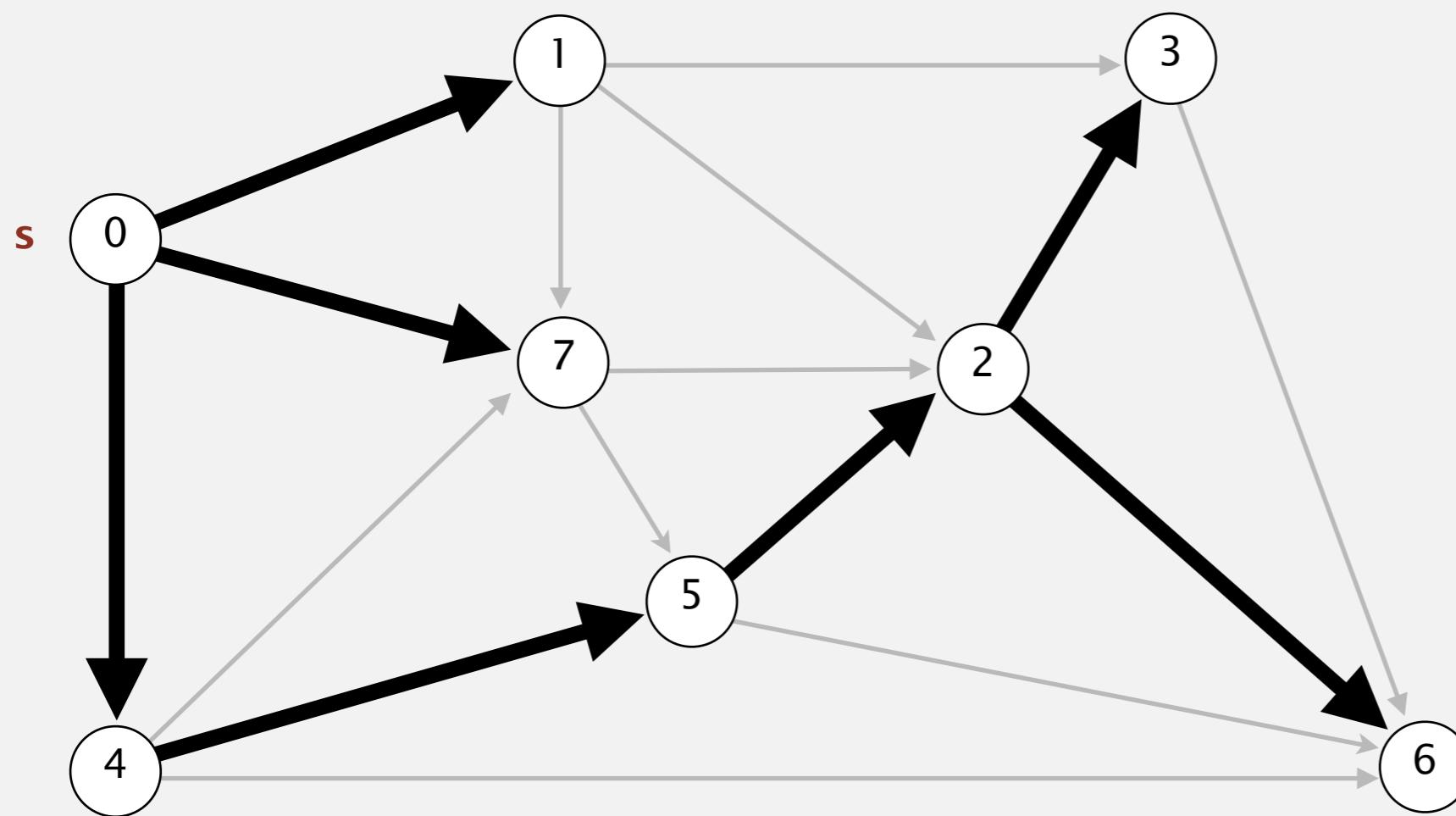
- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.

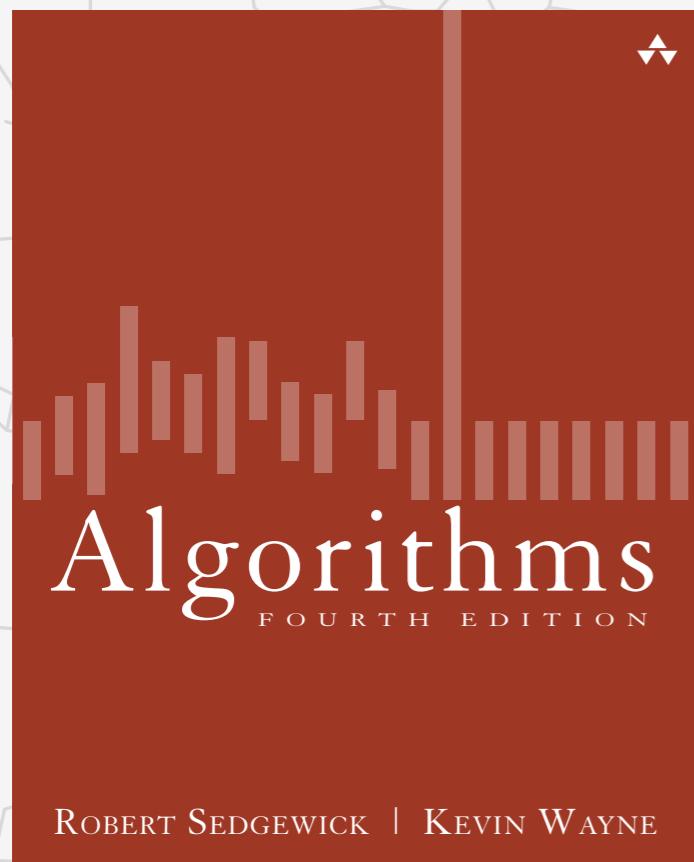


shortest-paths tree from vertex s

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

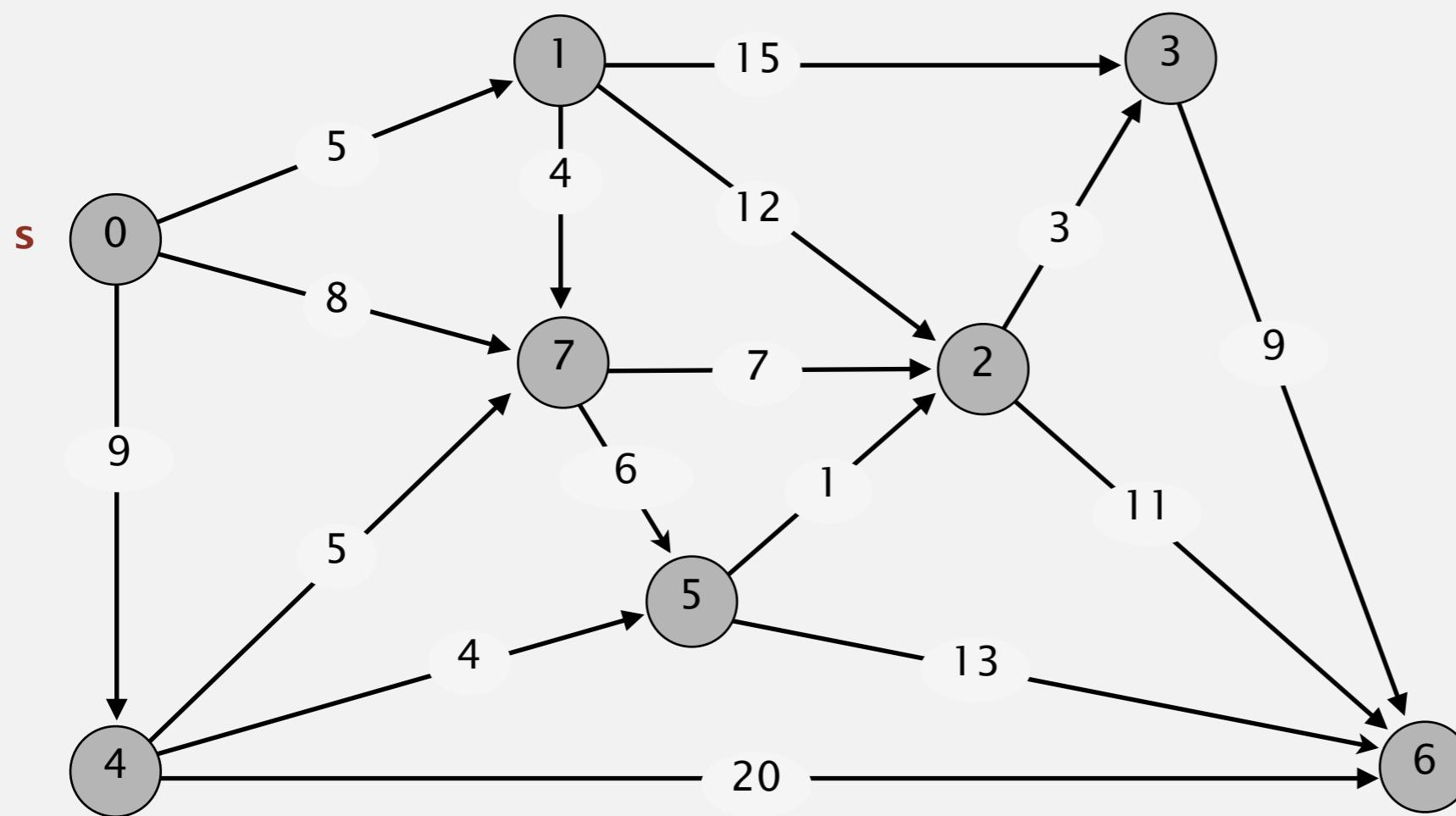


BELLMAN-FORD DEMO

<http://algs4.cs.princeton.edu>

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.

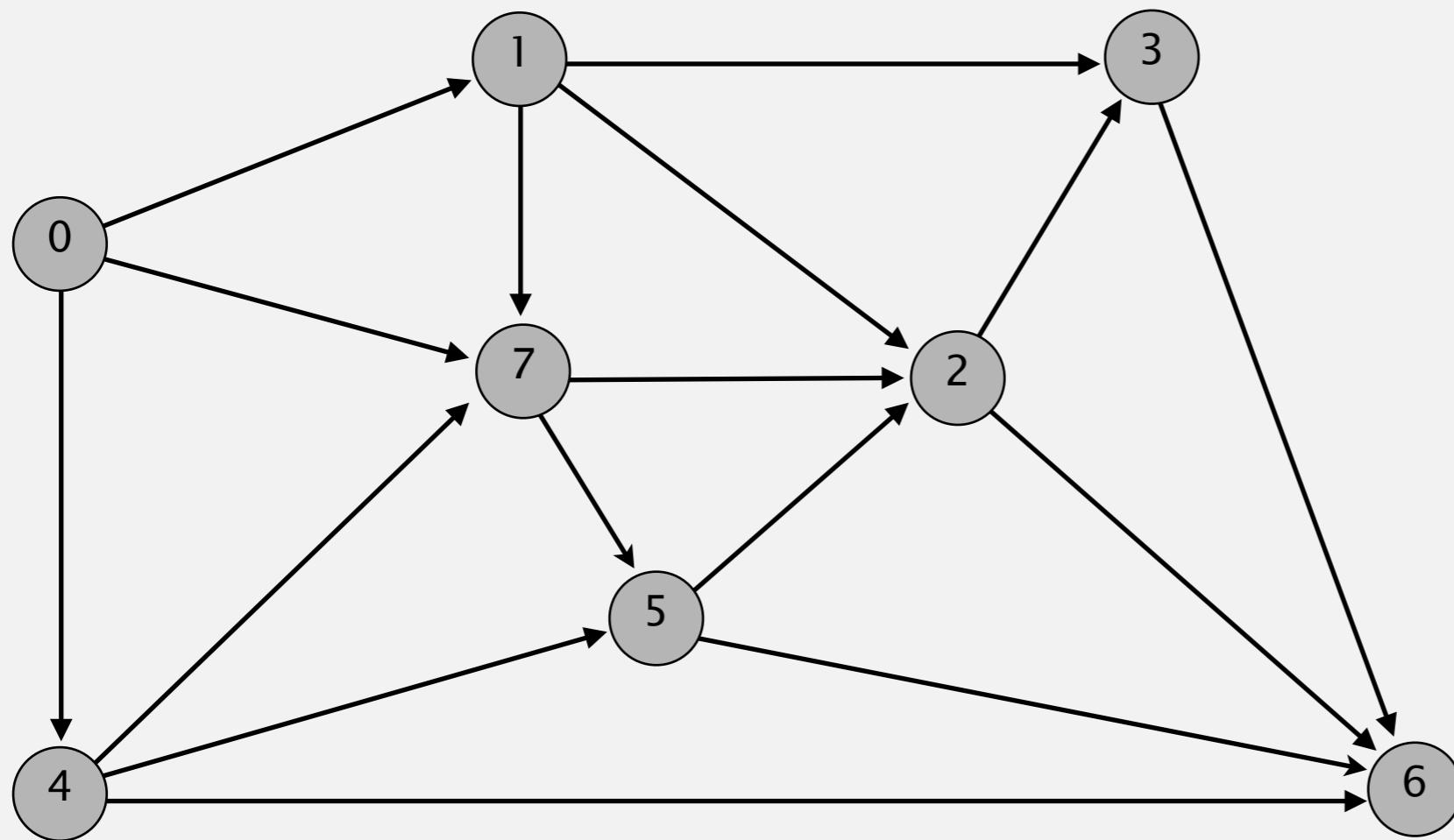


0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

an edge-weighted digraph

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.

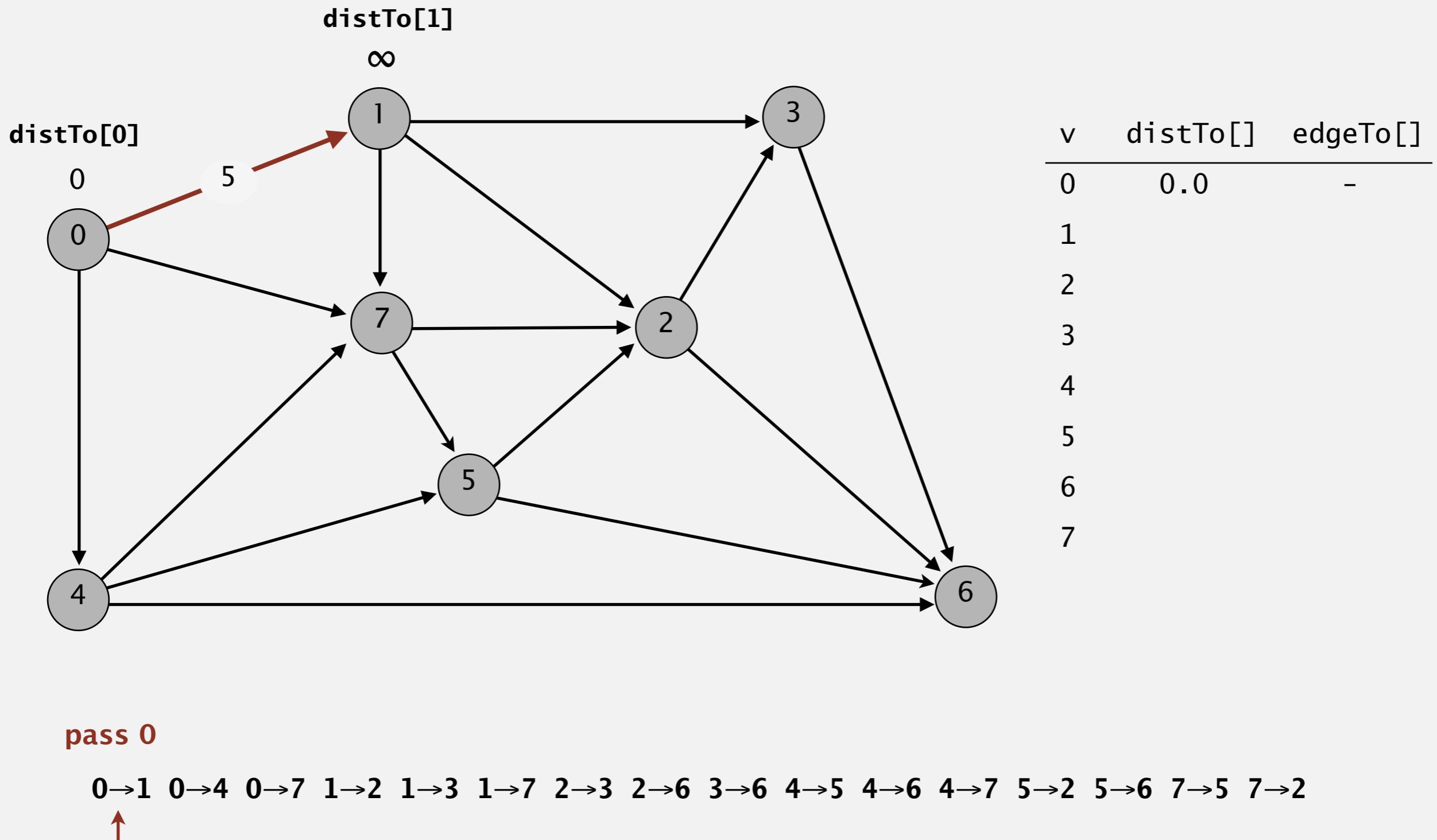


v	distTo[]	edgeTo[]
0	0.0	-
1		
2		
3		
4		
5		
6		
7		

initialize

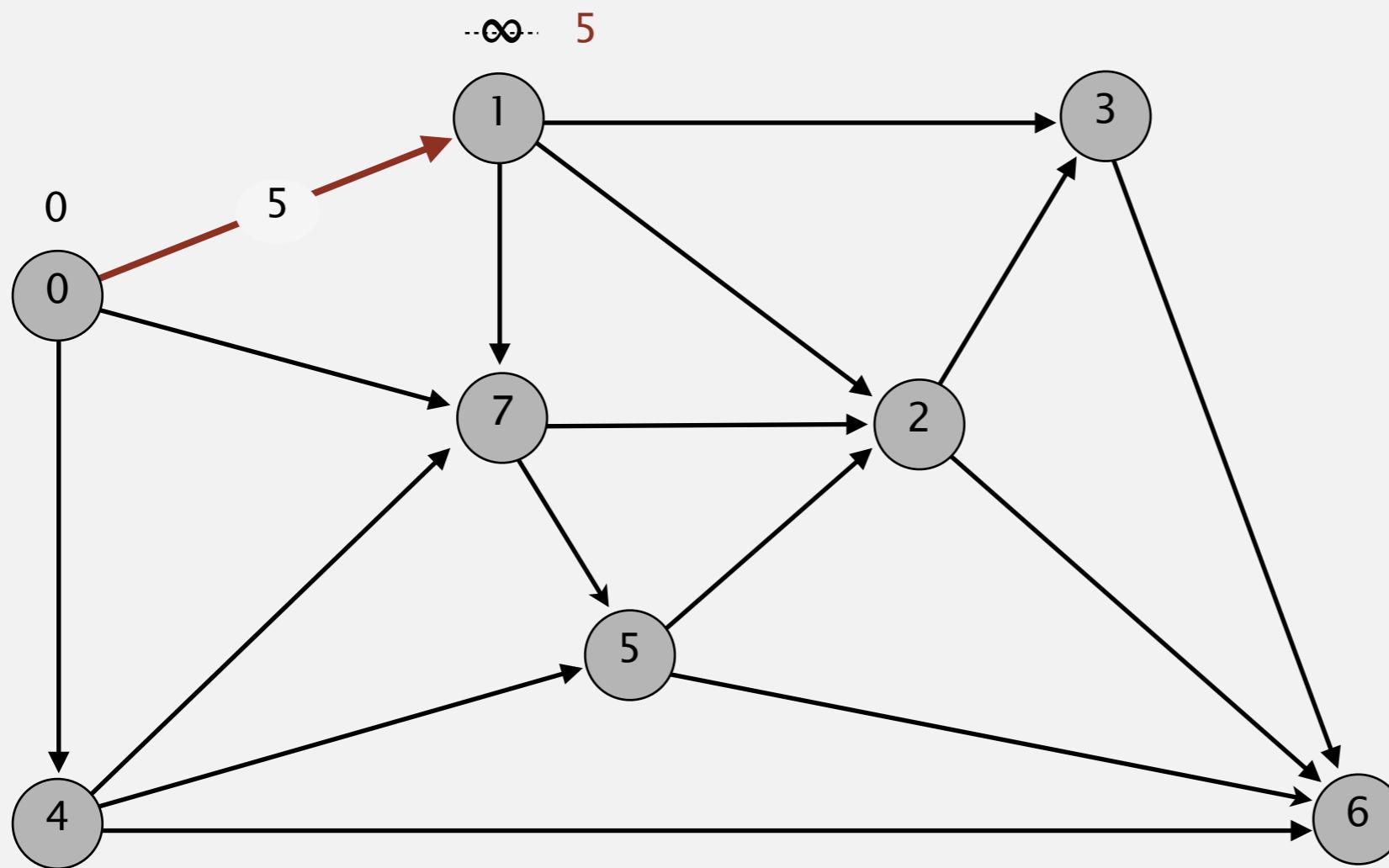
Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4		
5		
6		
7		

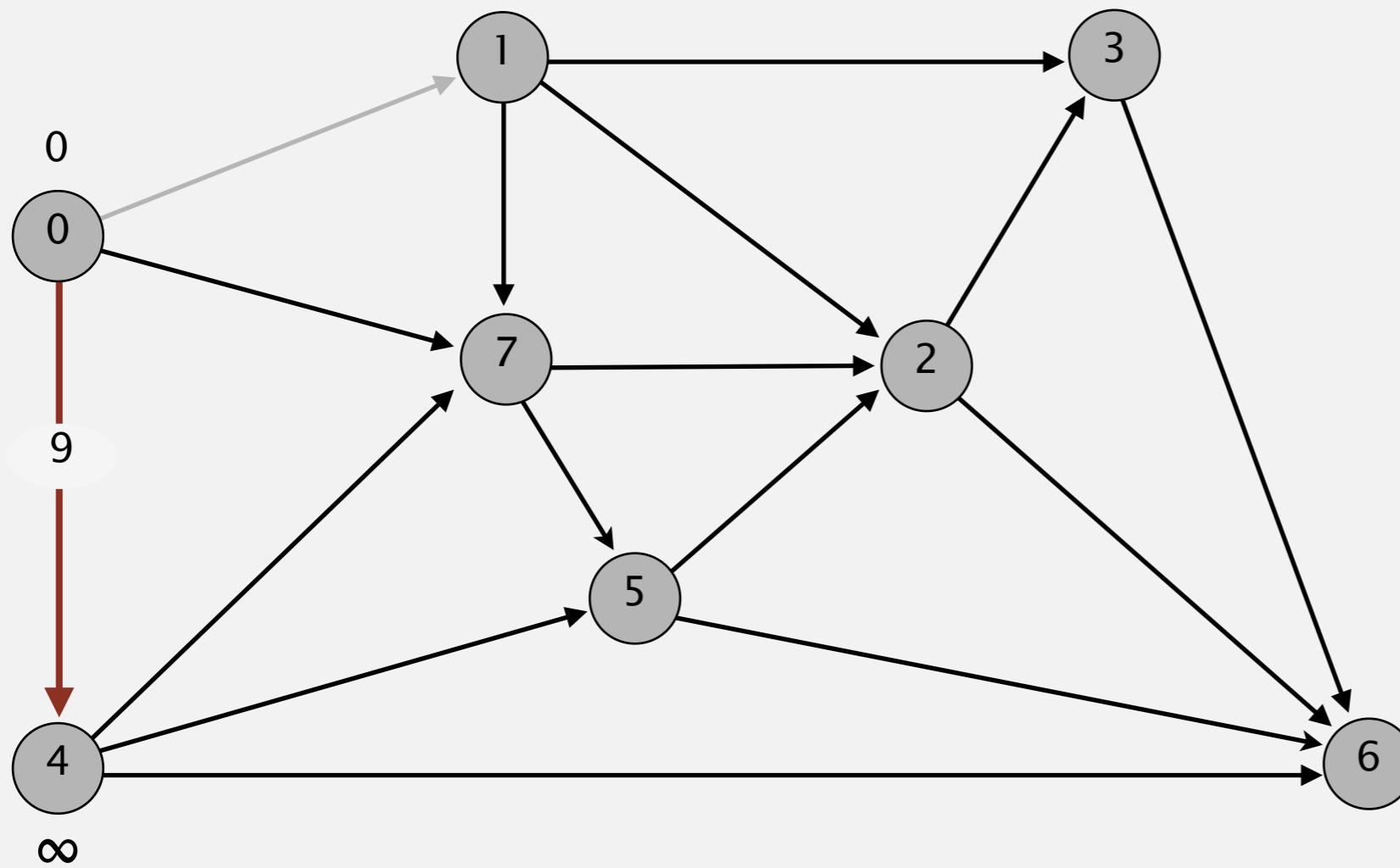
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4		
5		
6		
7		

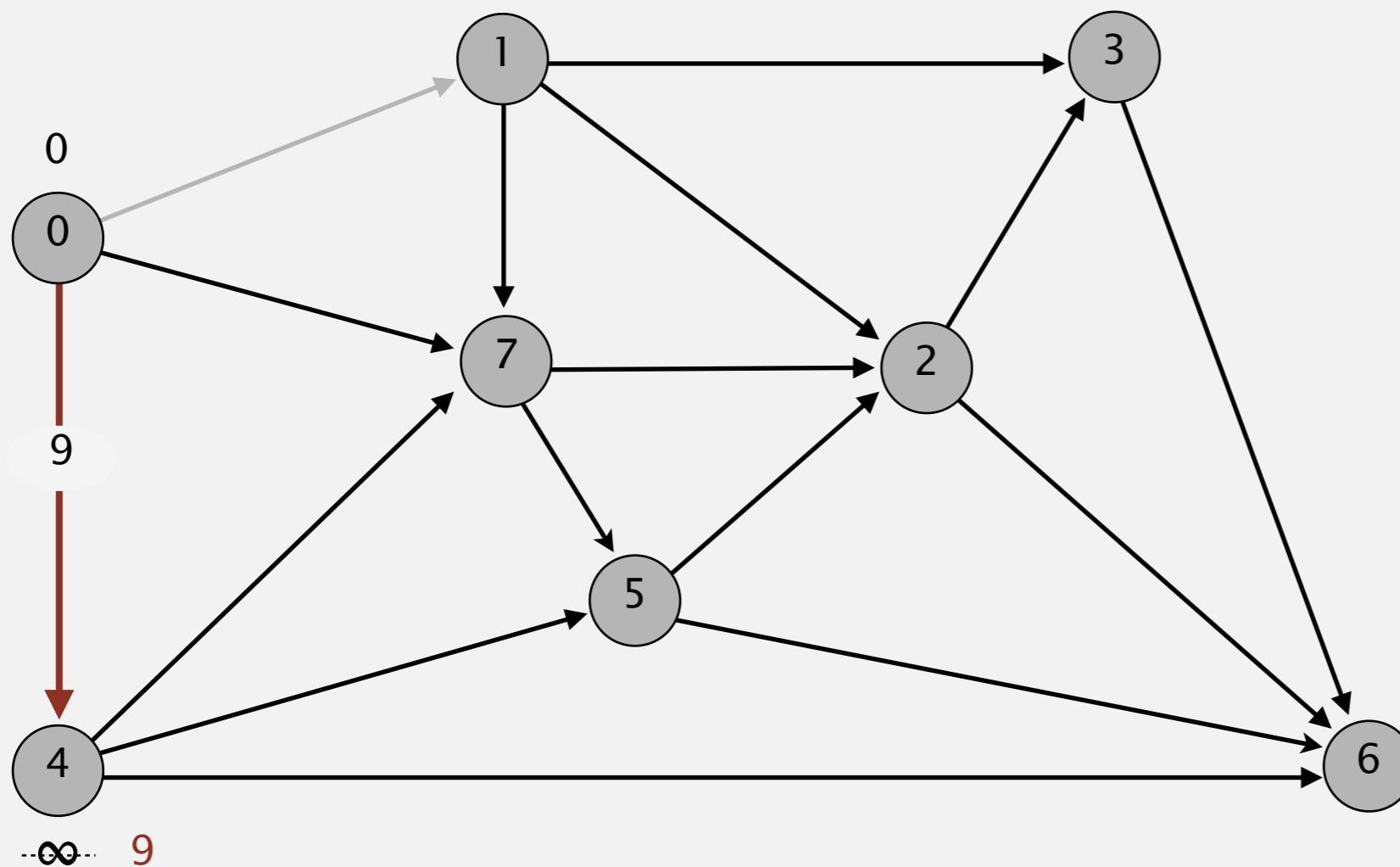
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



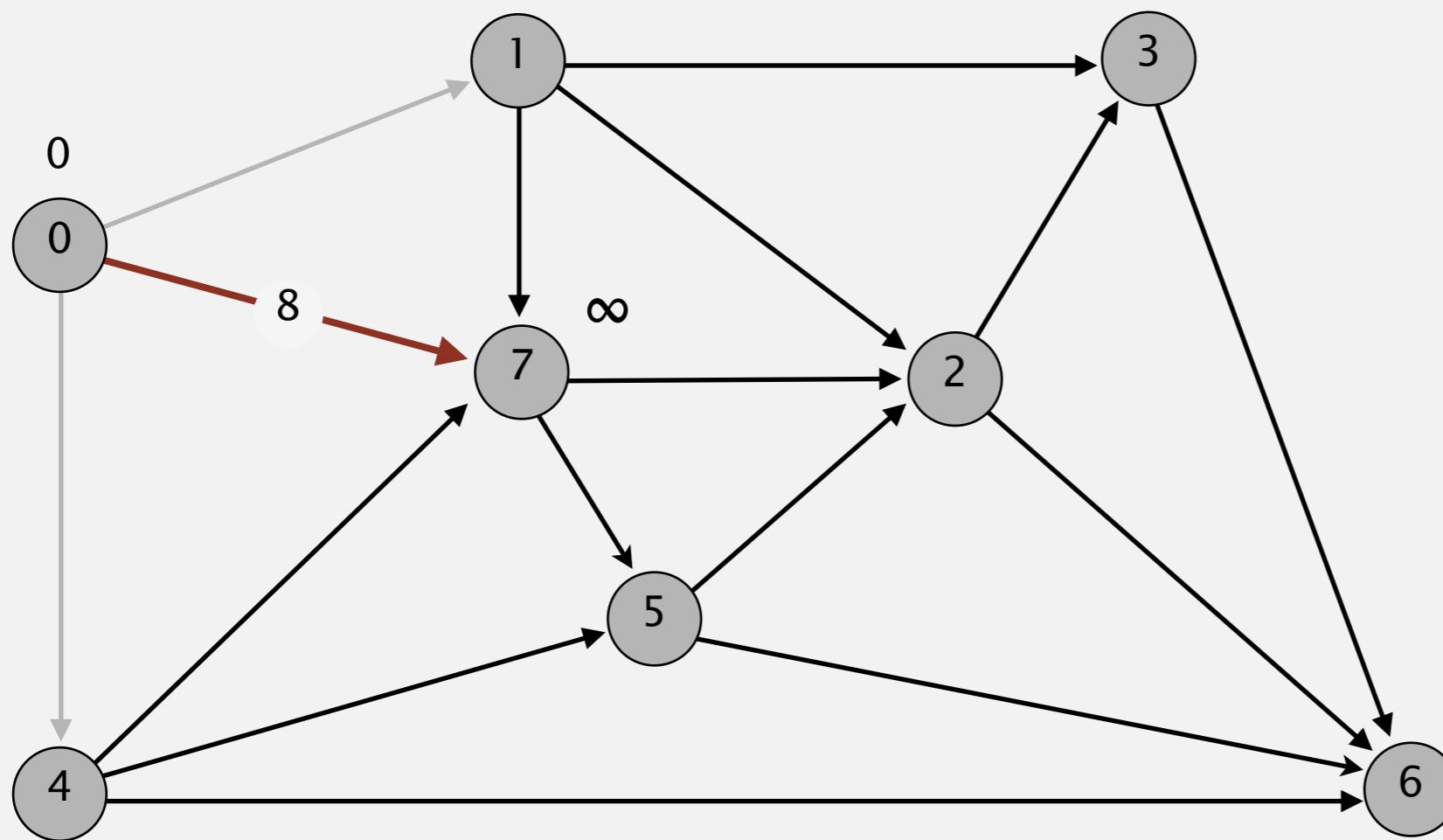
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7		

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



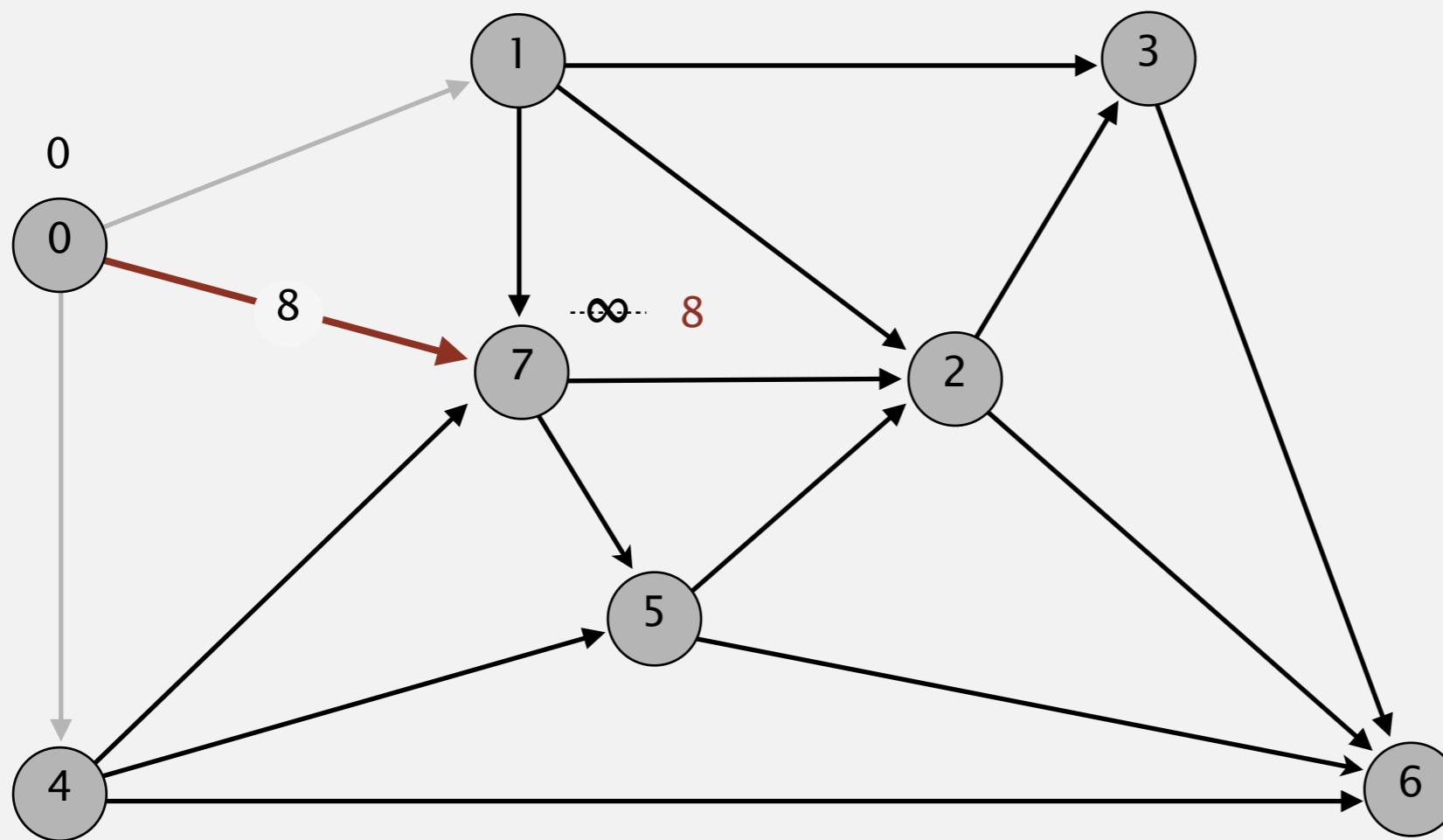
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7		

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

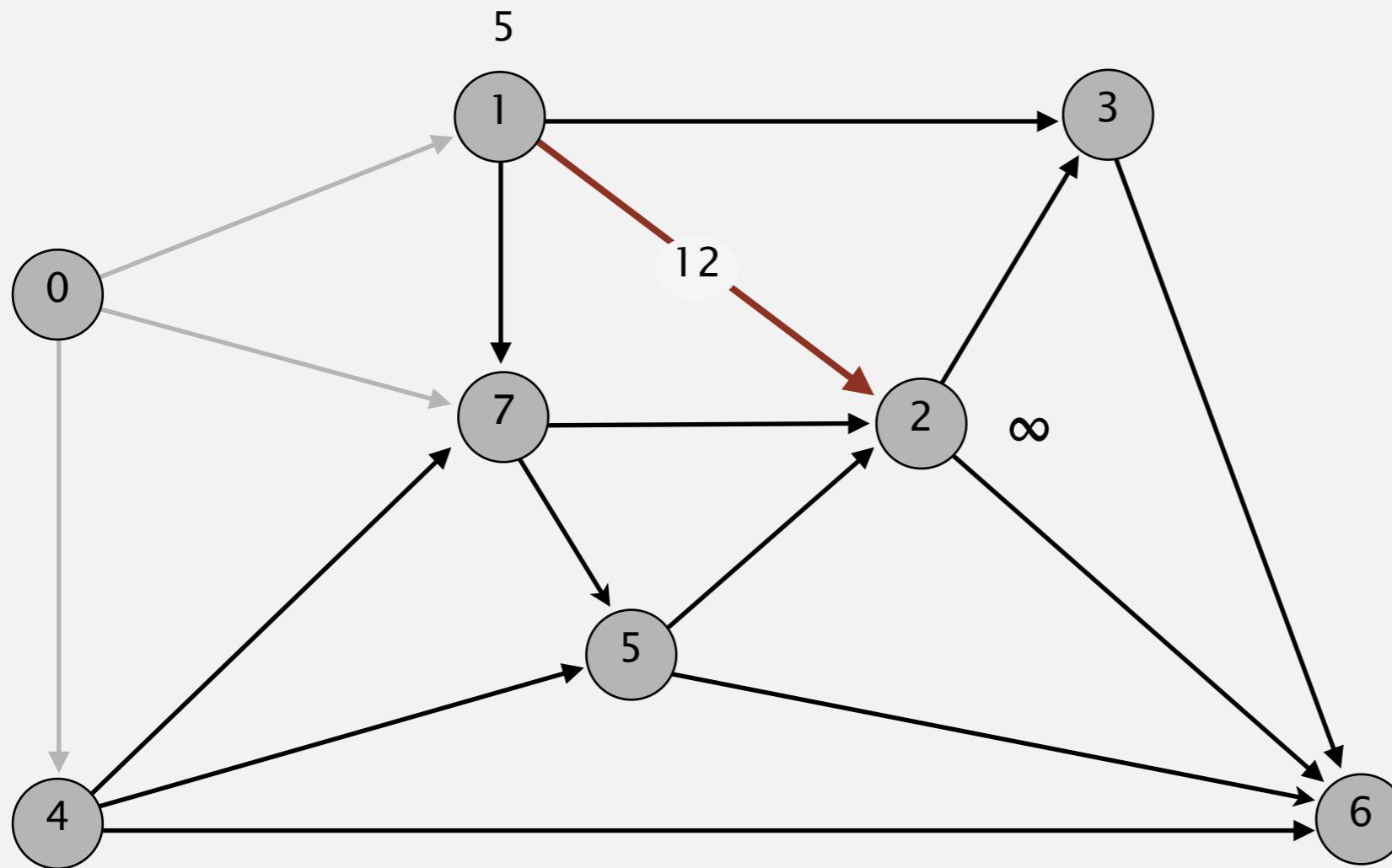
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

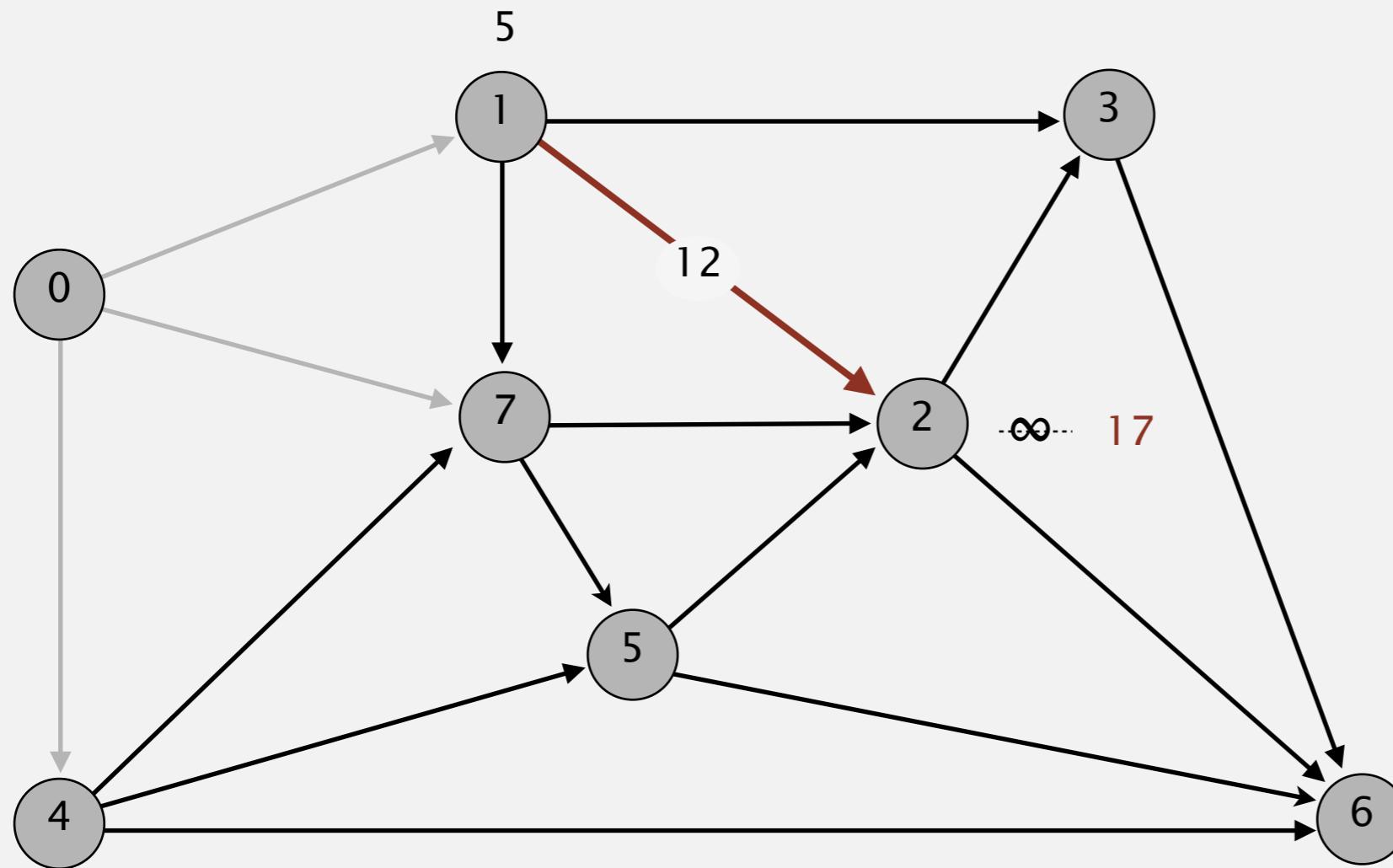
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

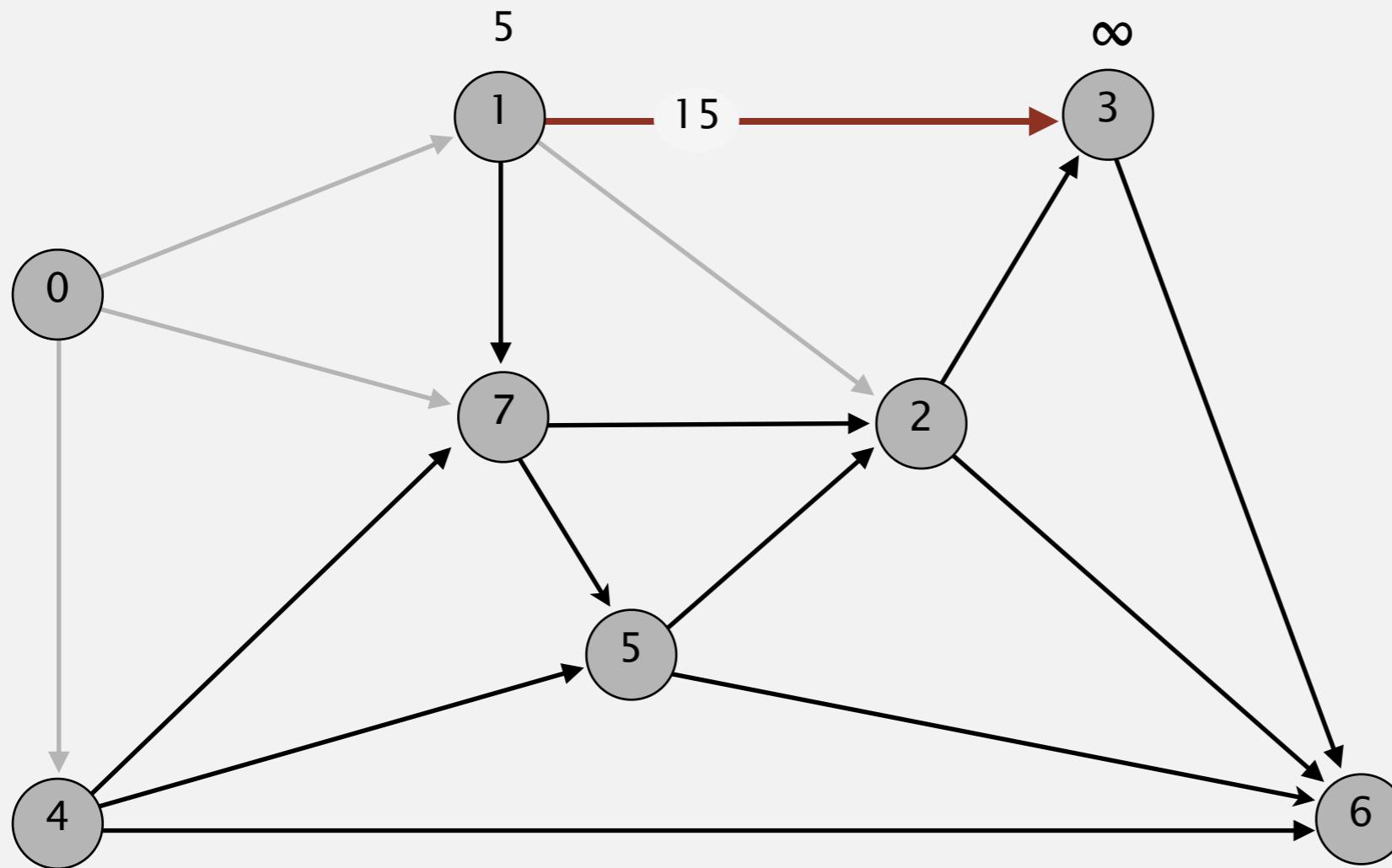
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

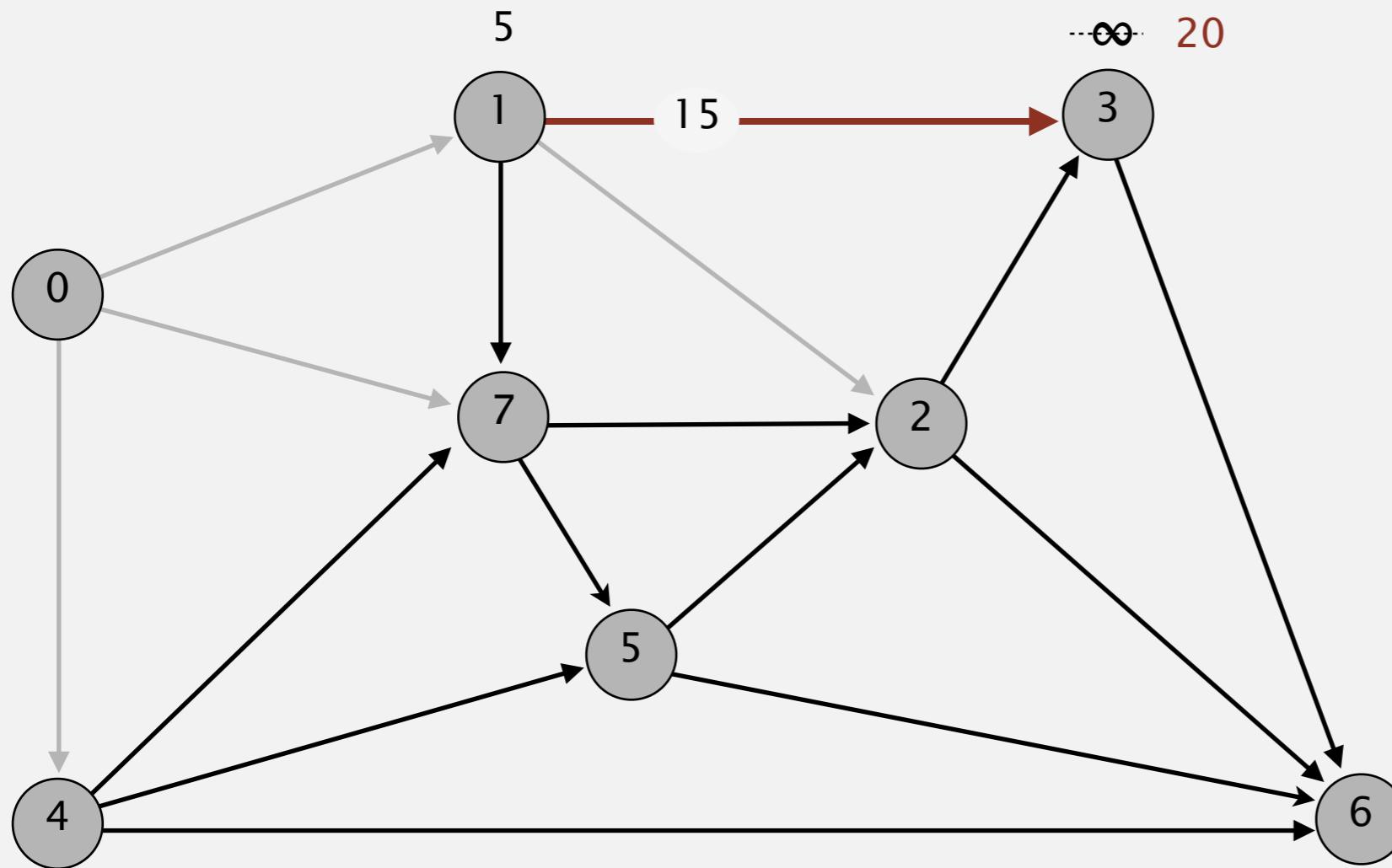
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

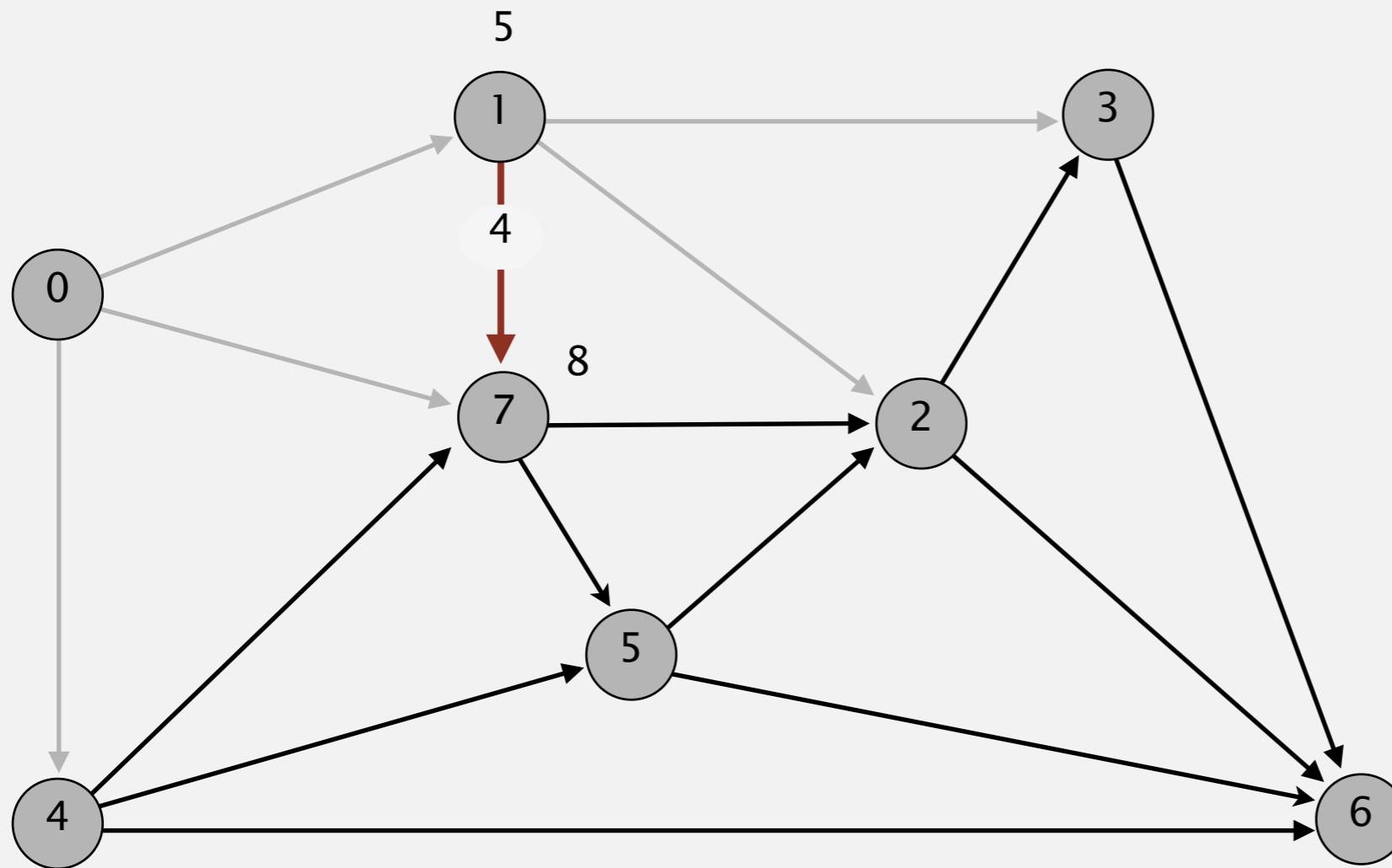
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

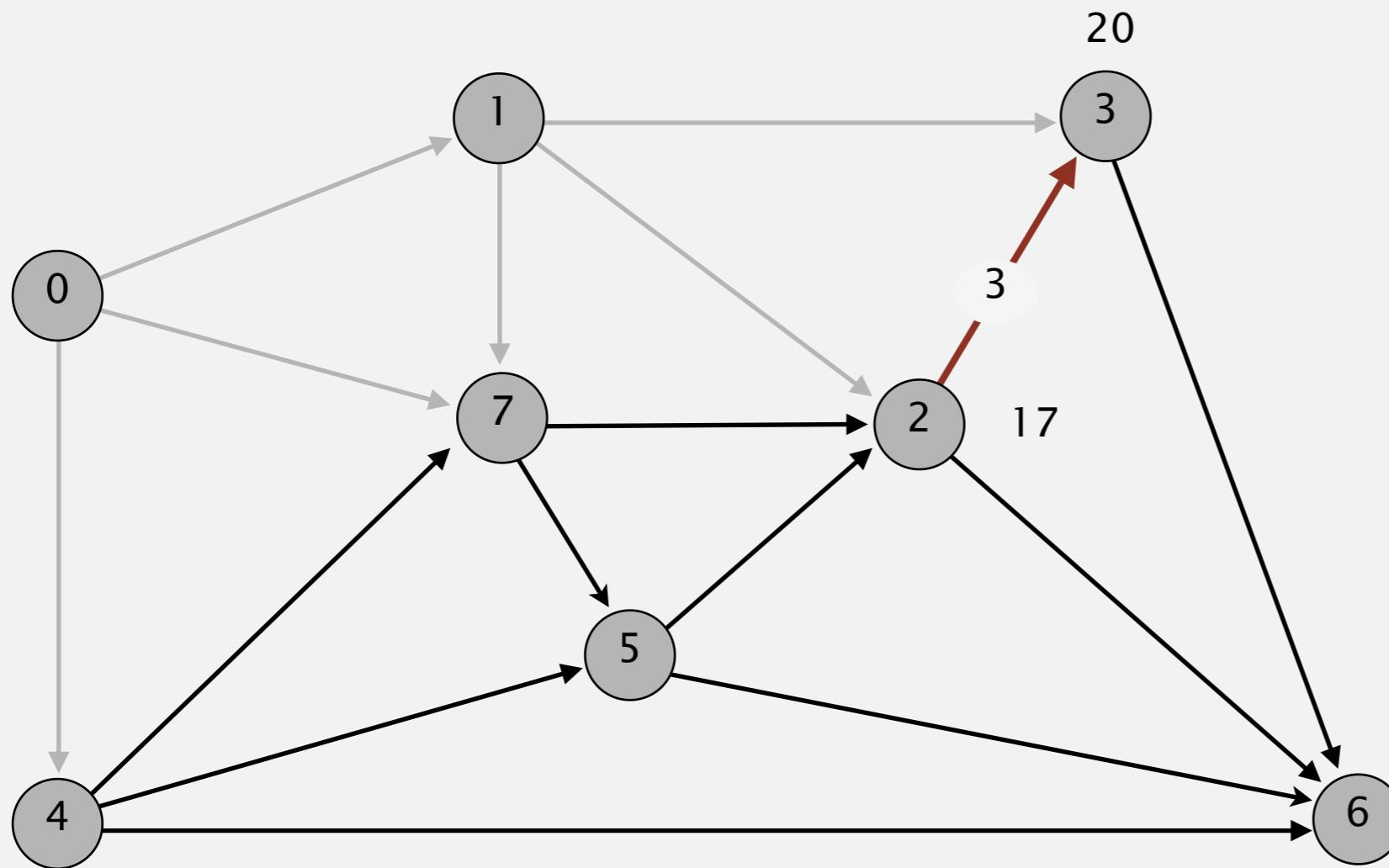
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

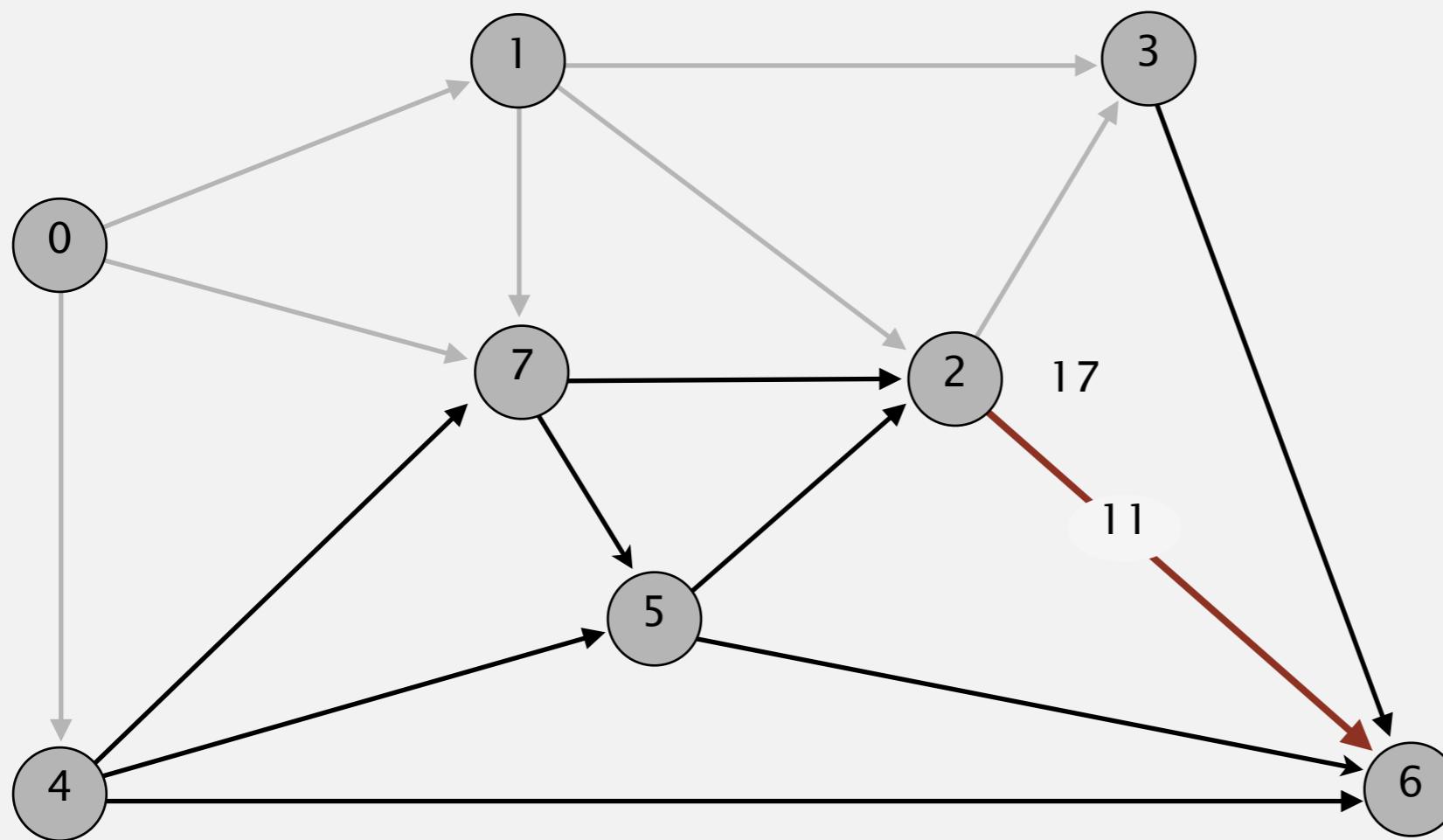
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7
∞		

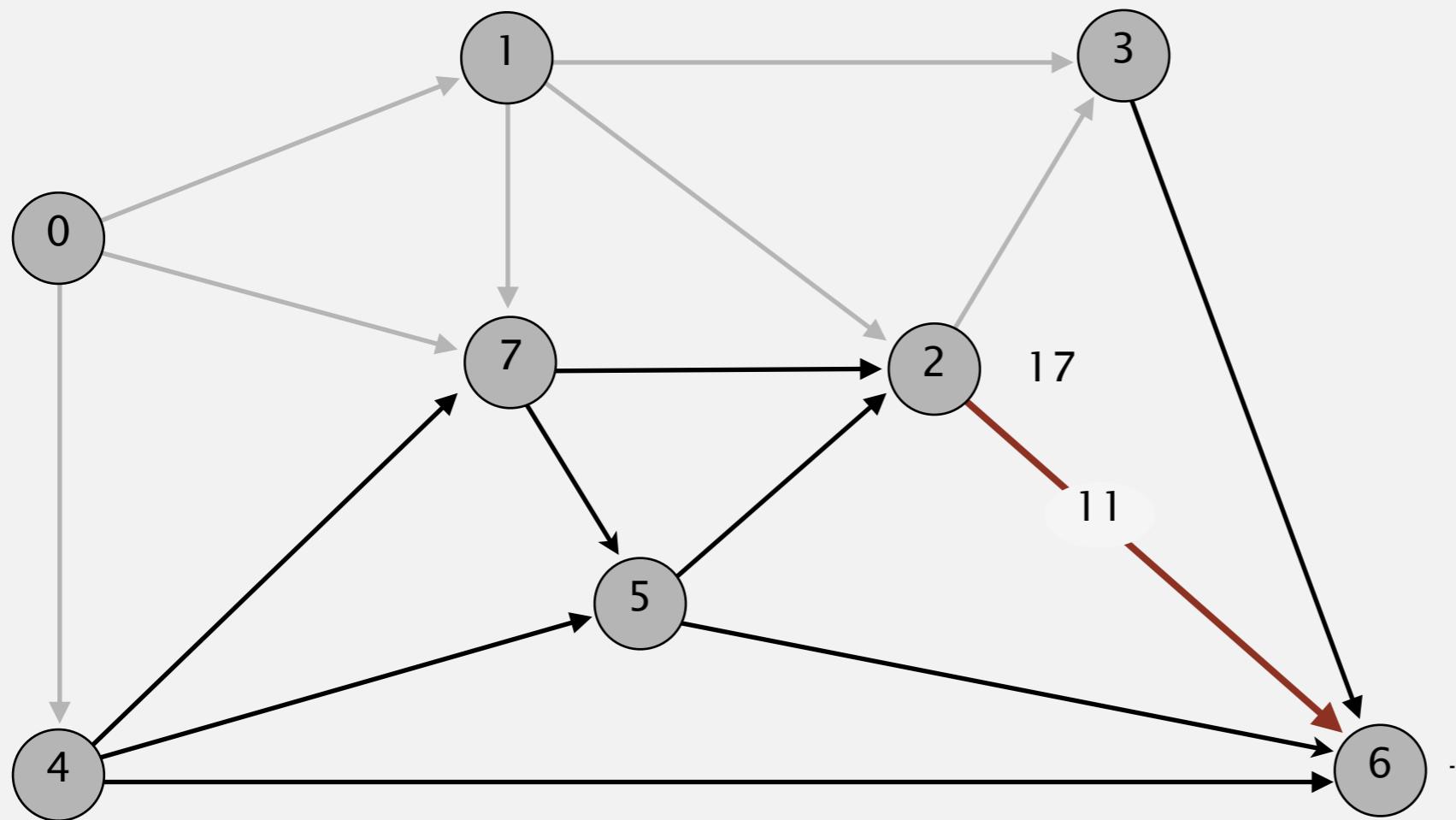
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6	28.0	2→6
7	8.0	0→7

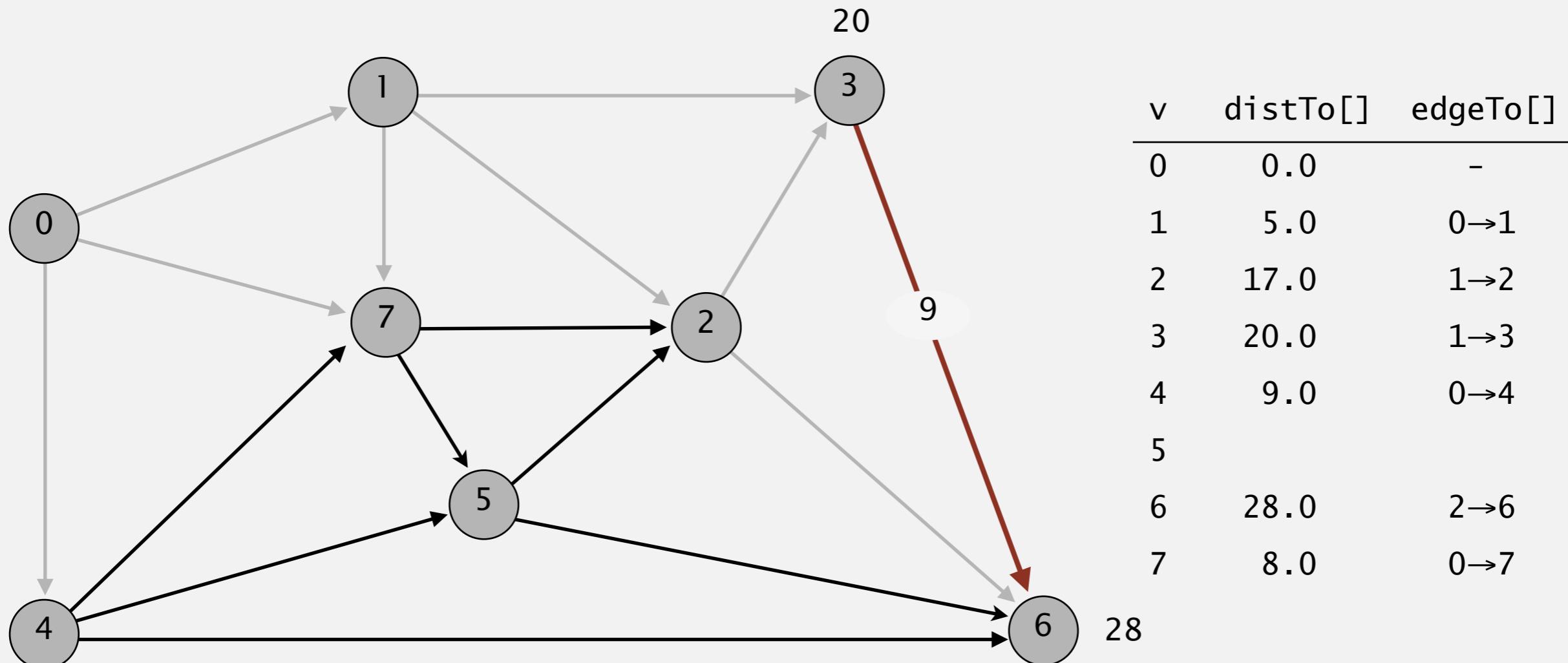
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.

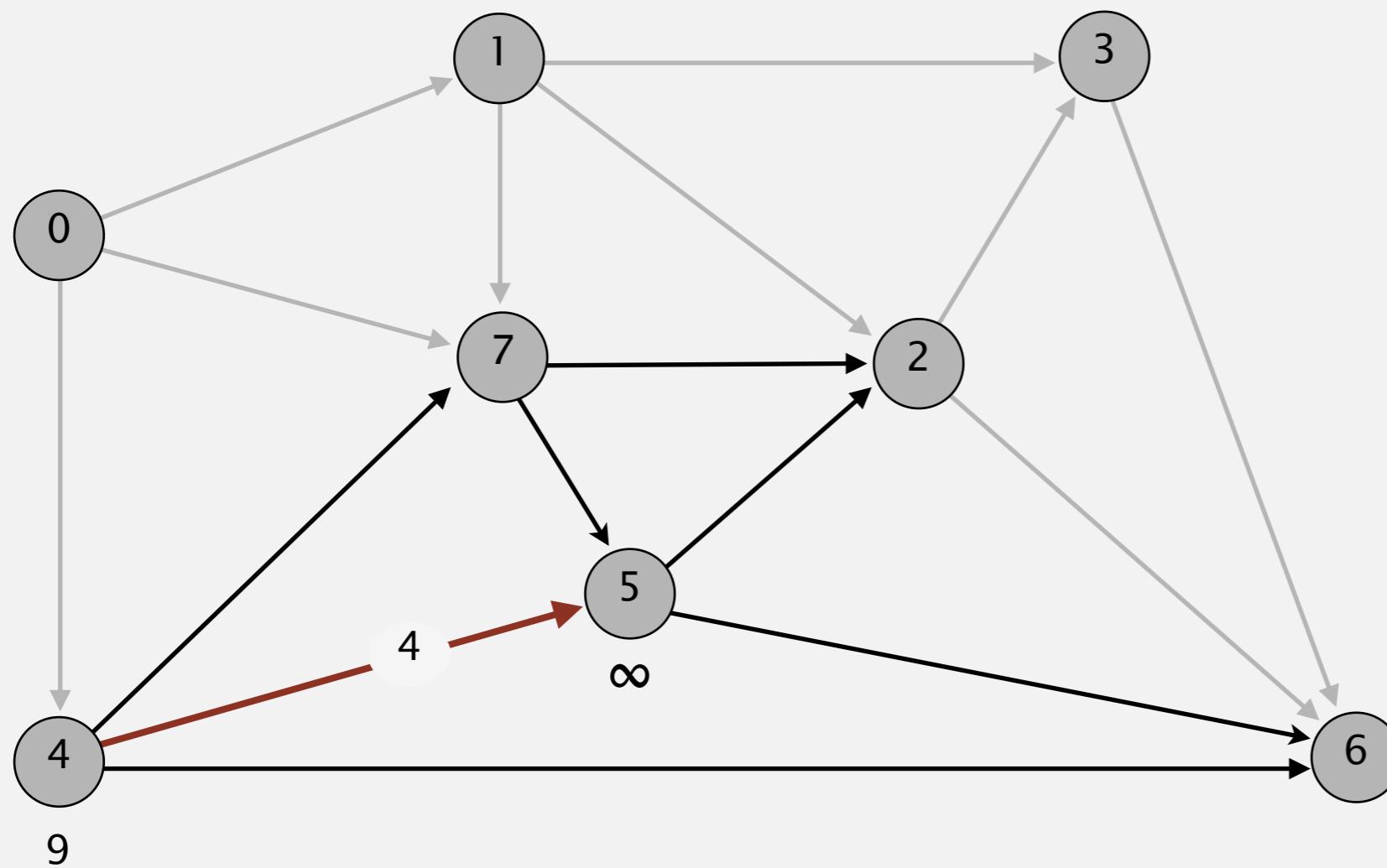


pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



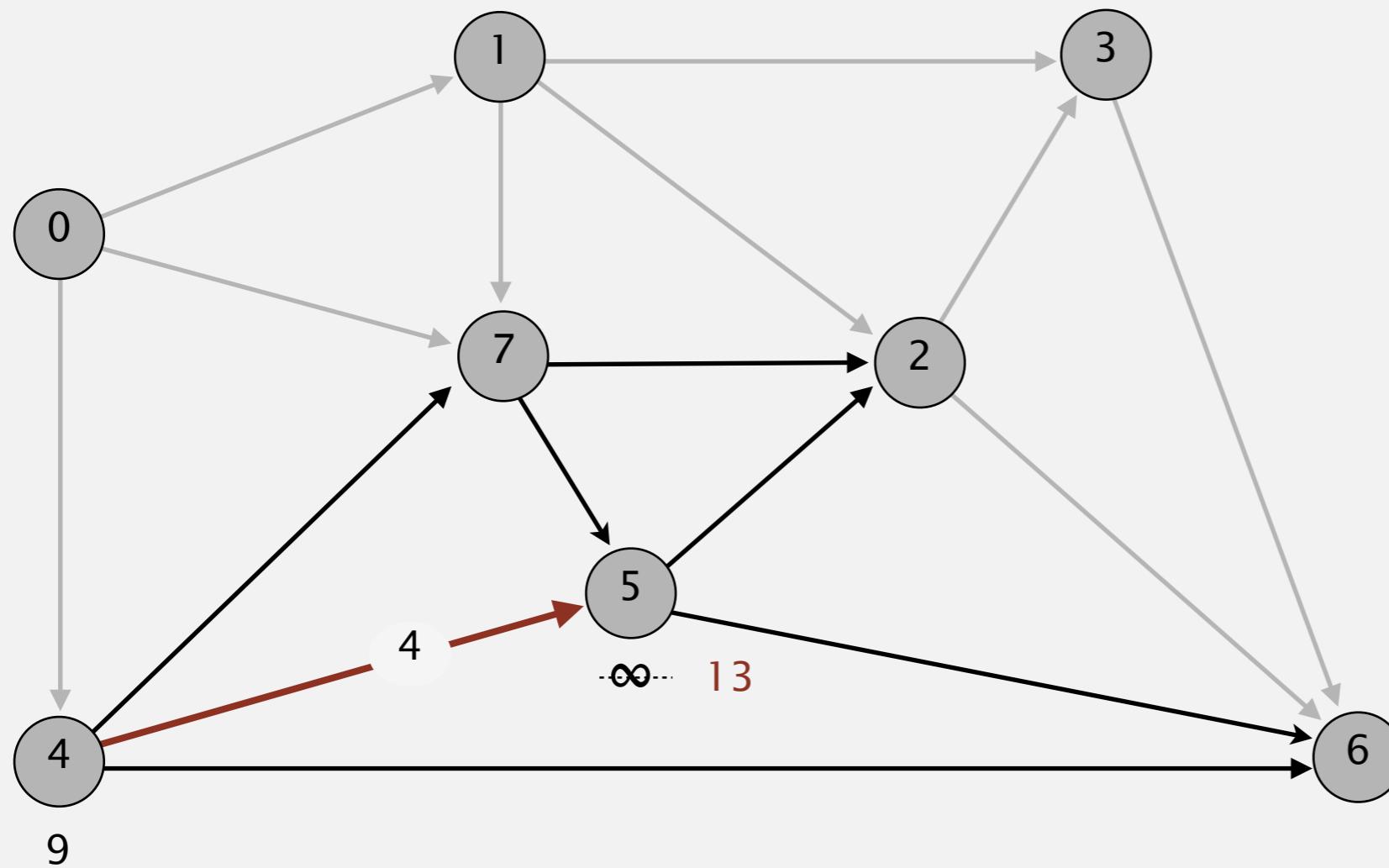
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



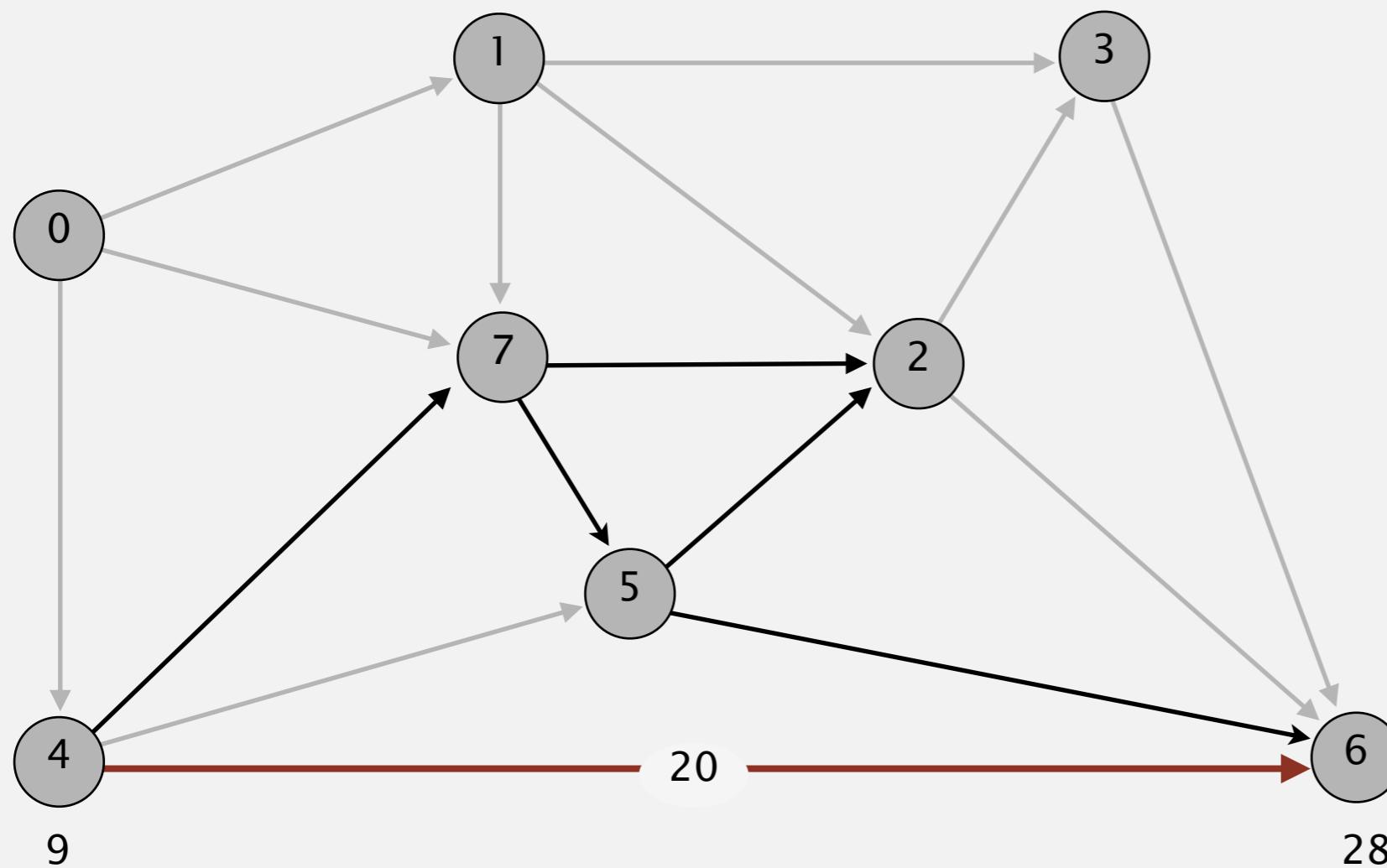
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



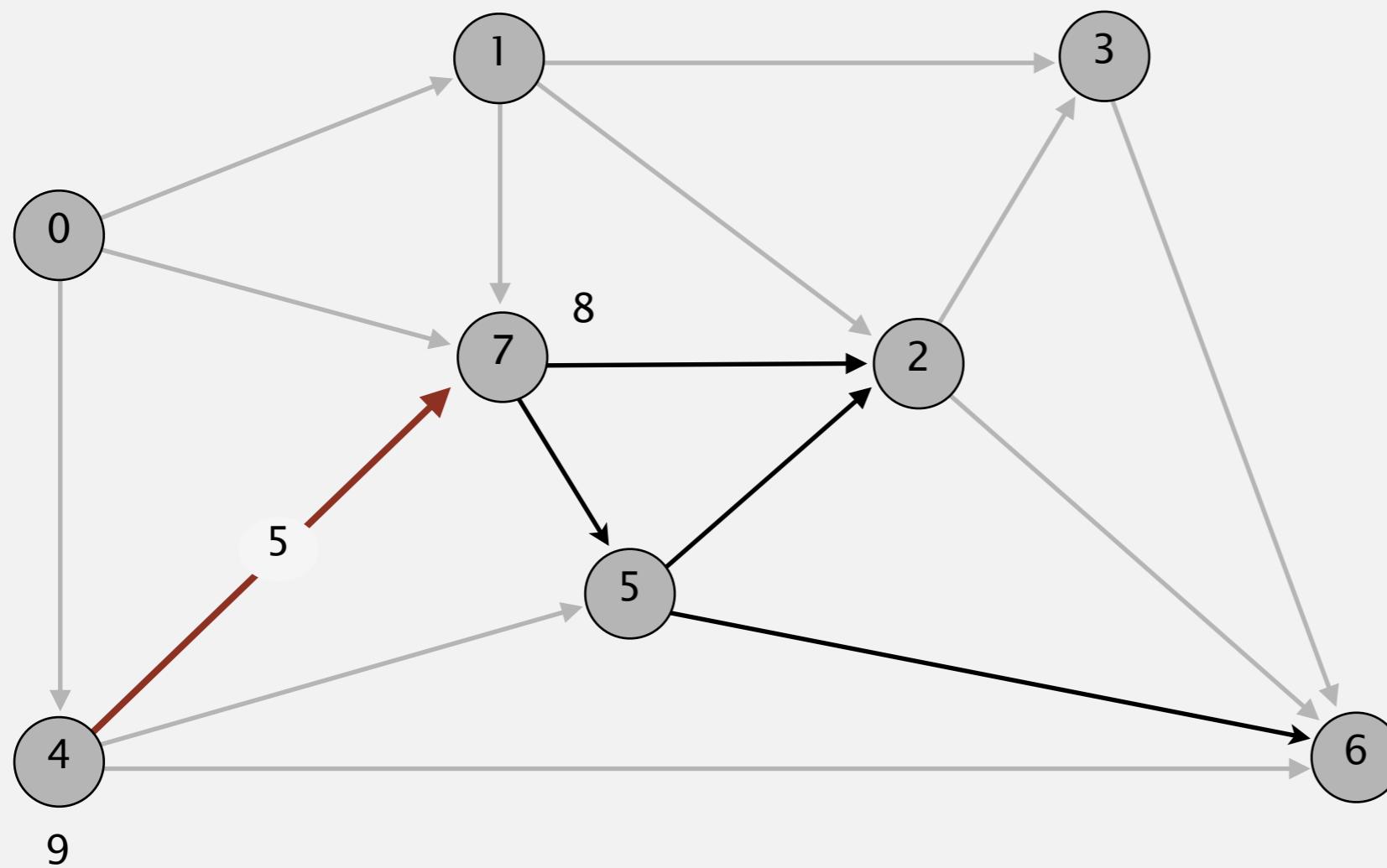
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



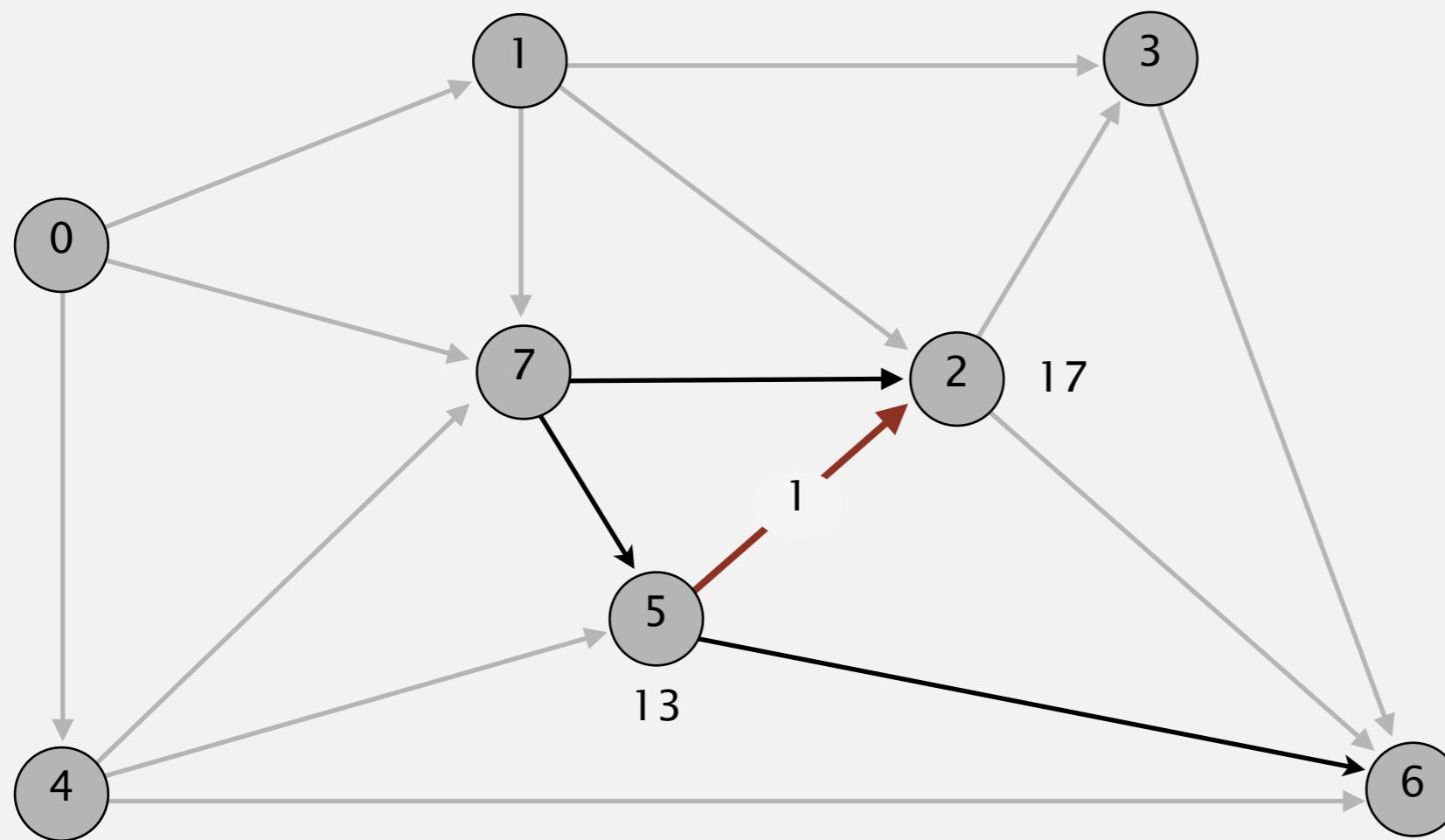
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



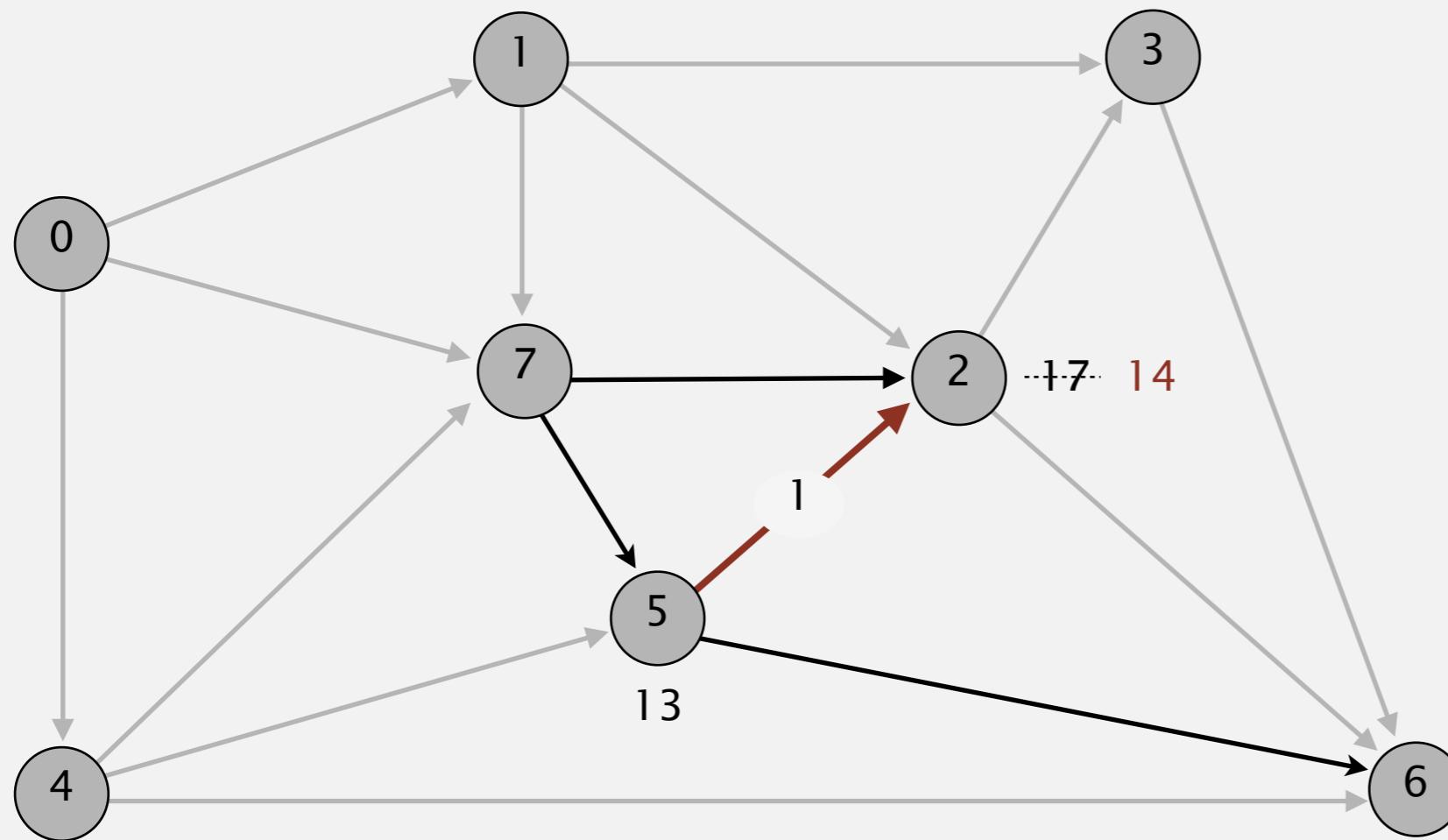
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



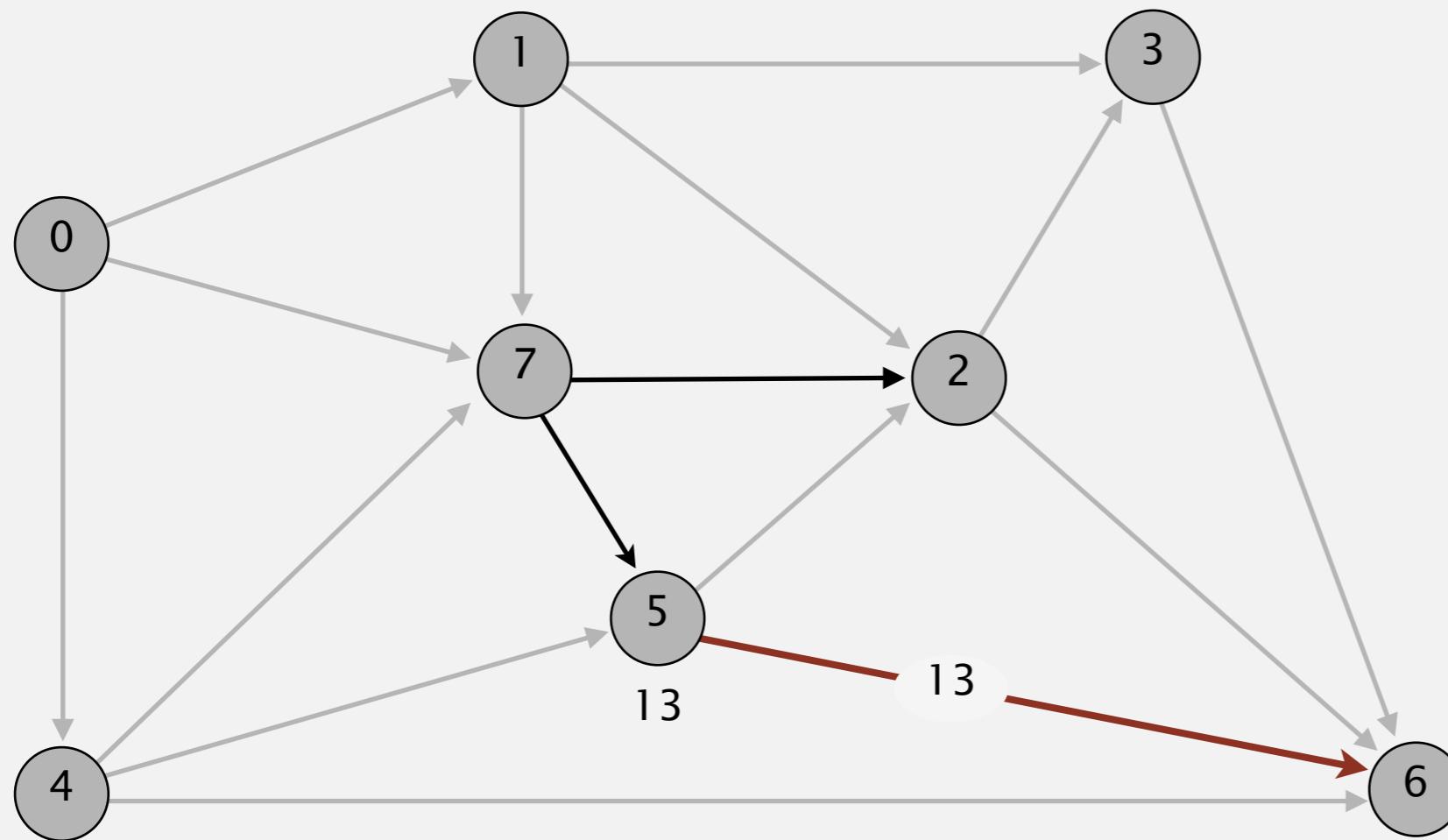
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

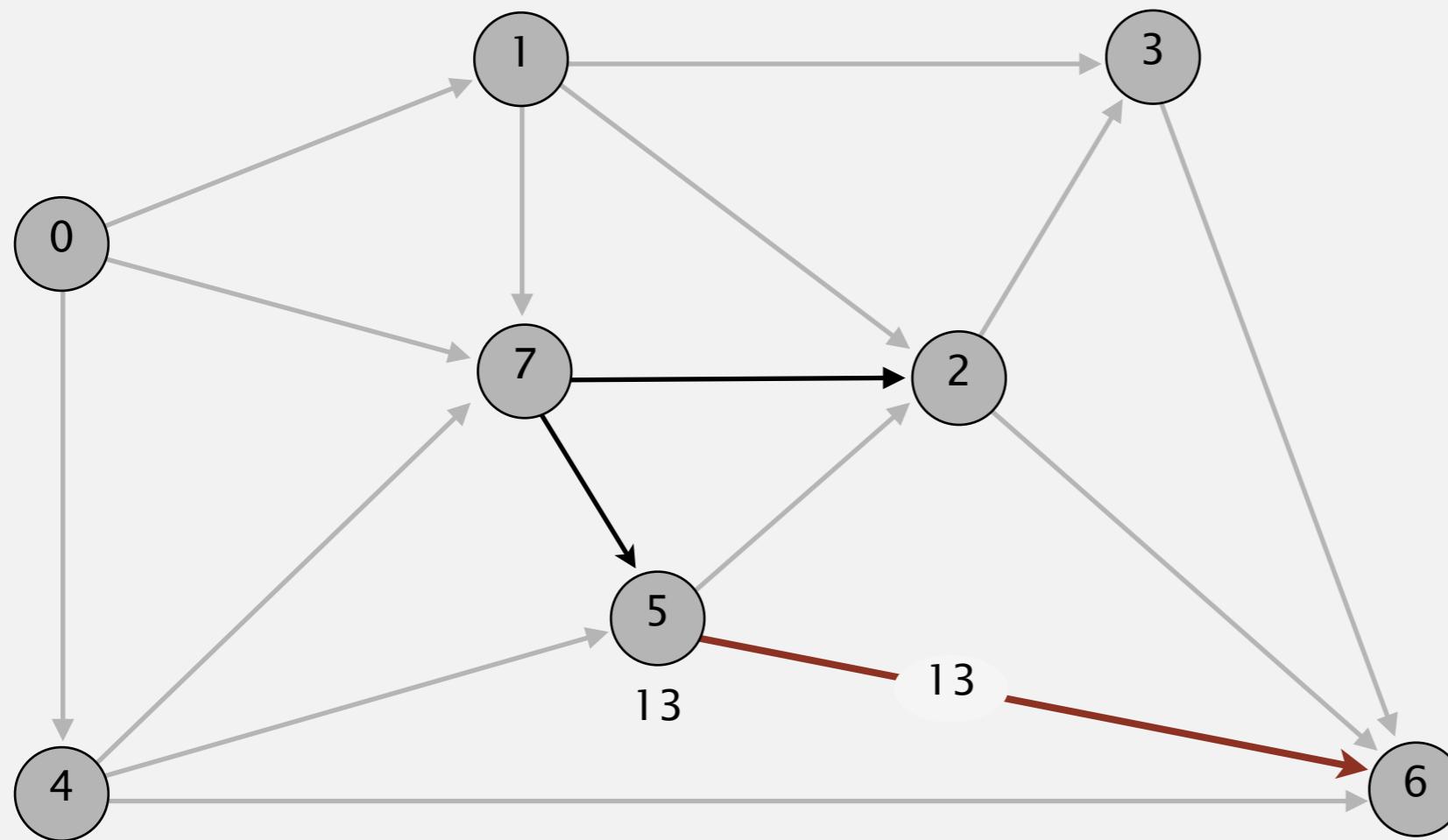
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

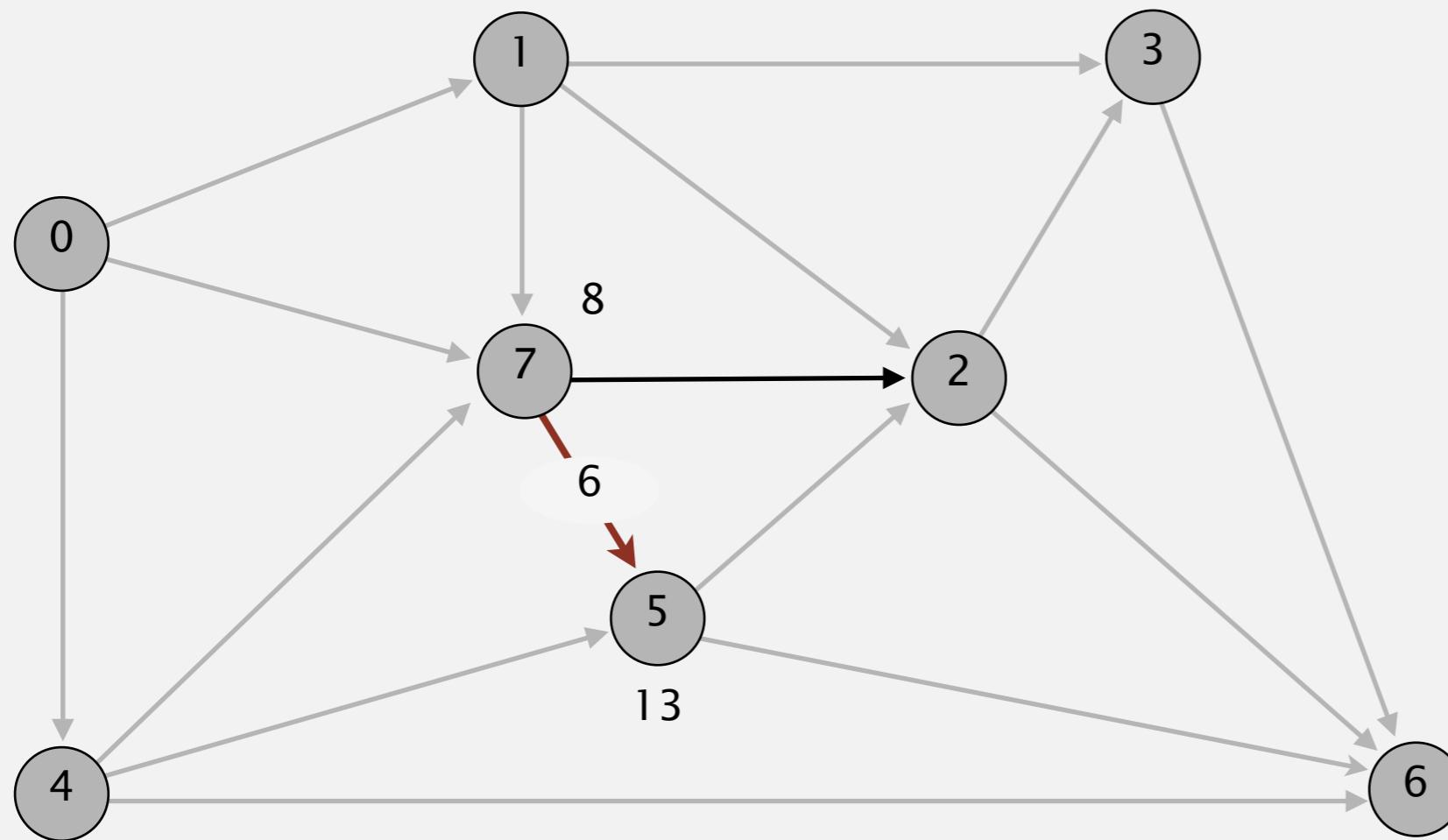
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

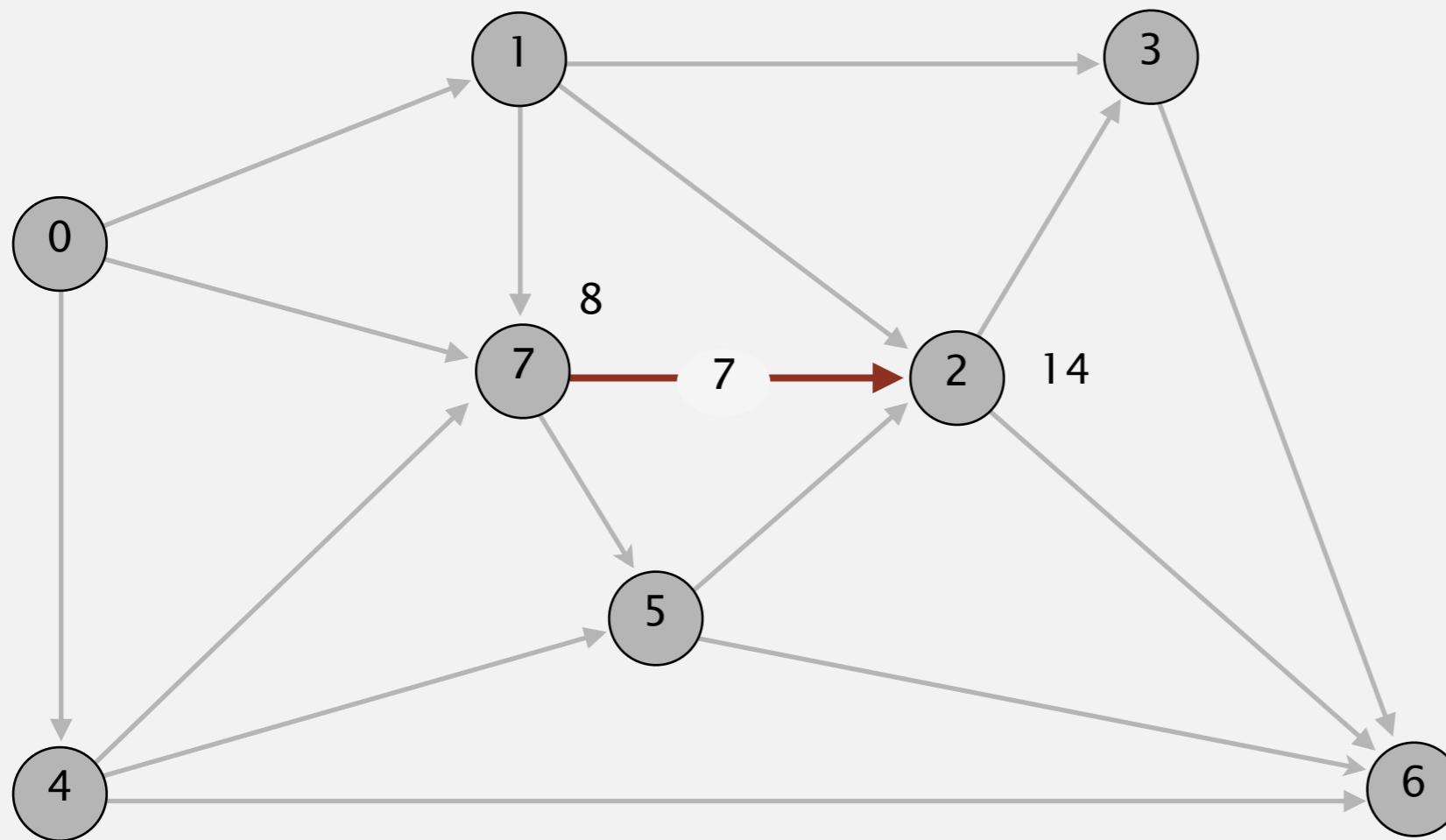
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

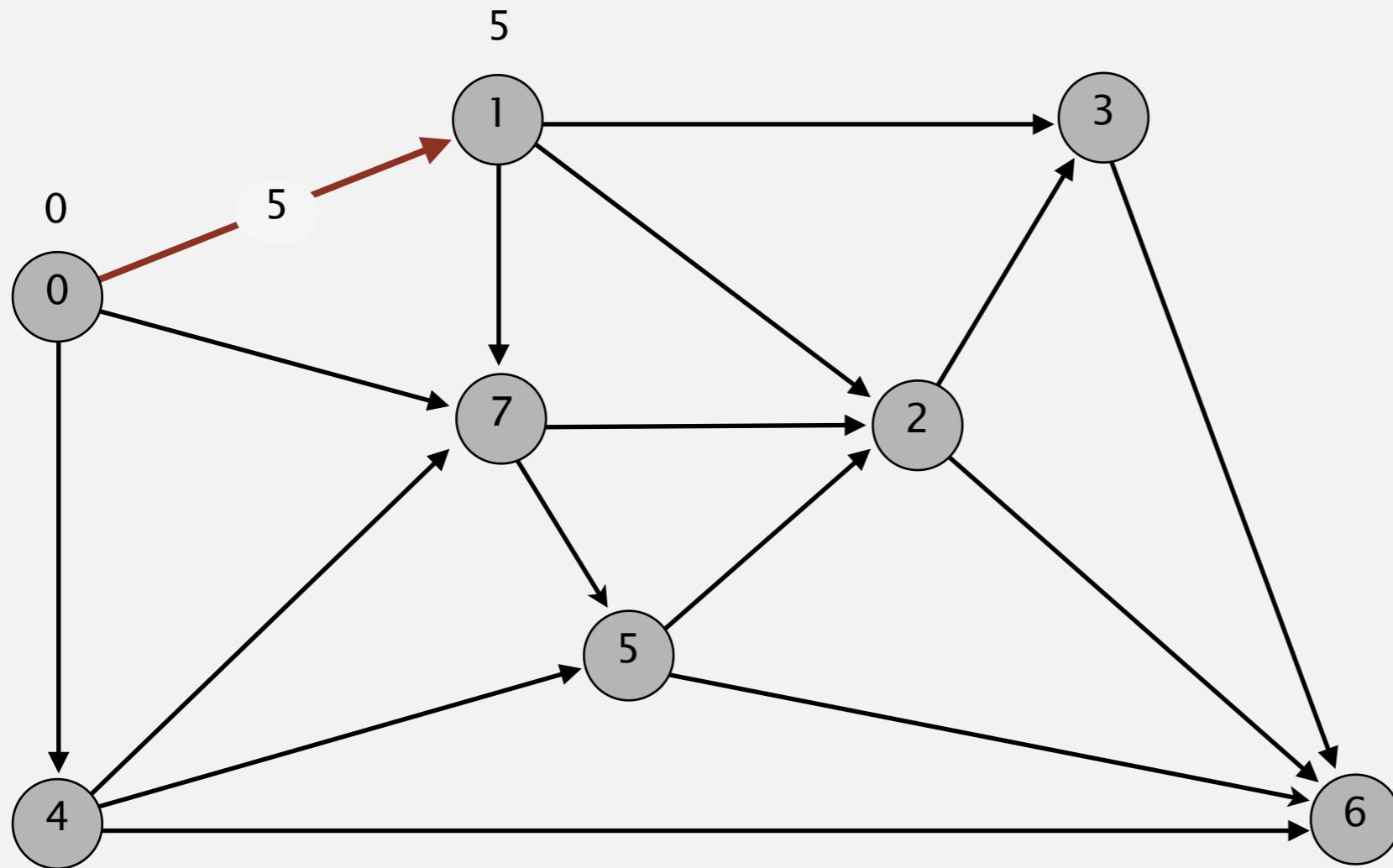
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

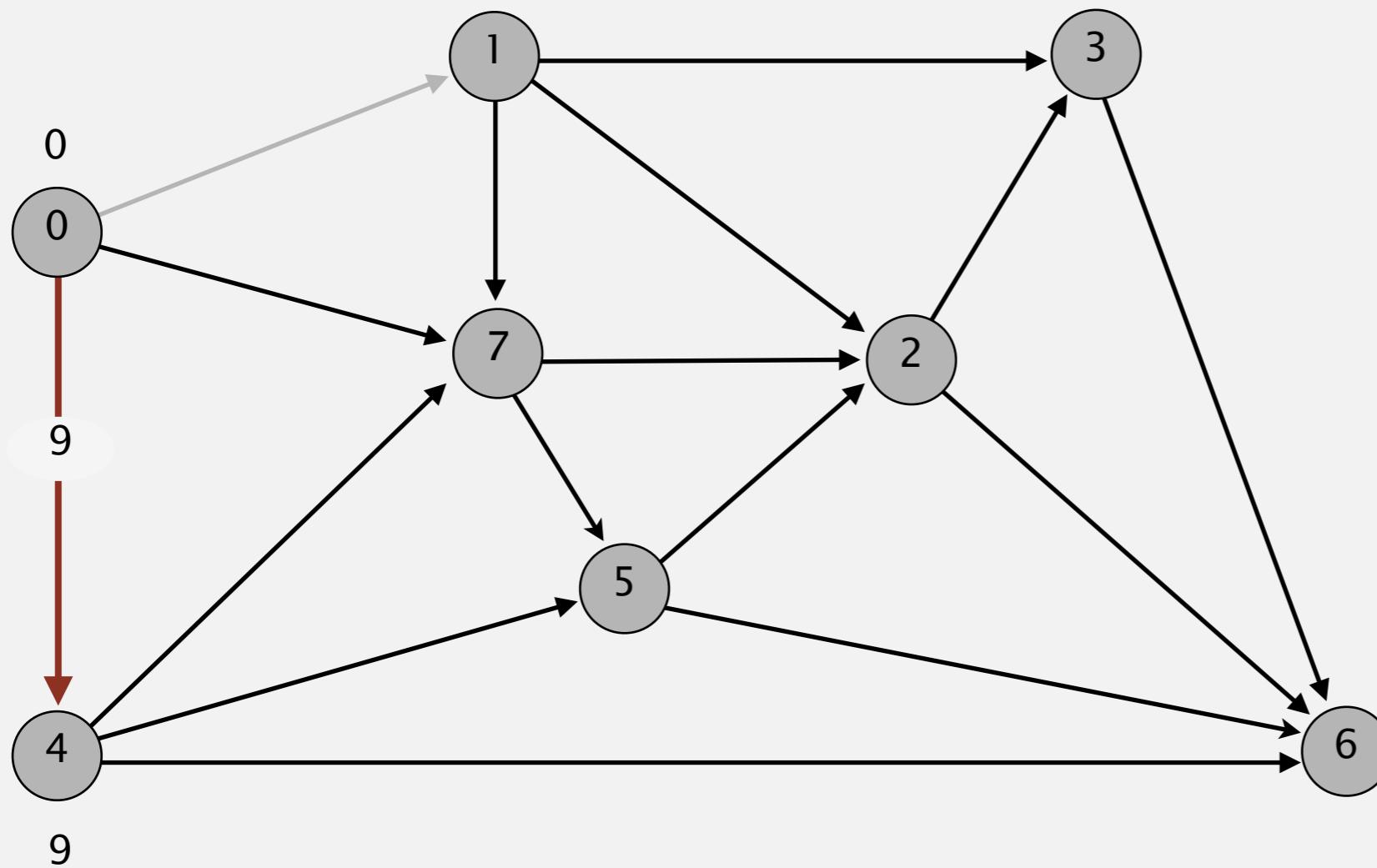
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

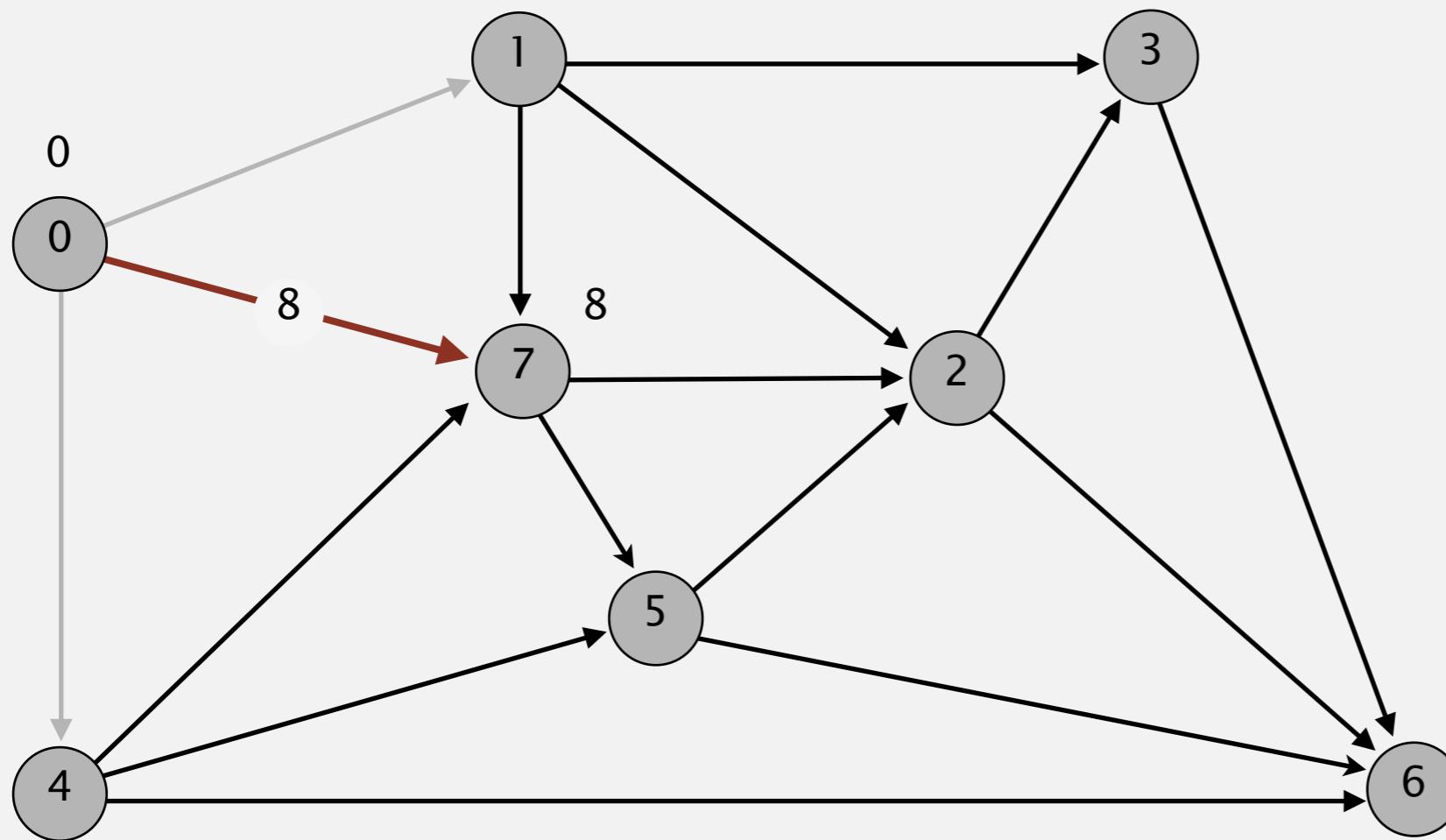
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

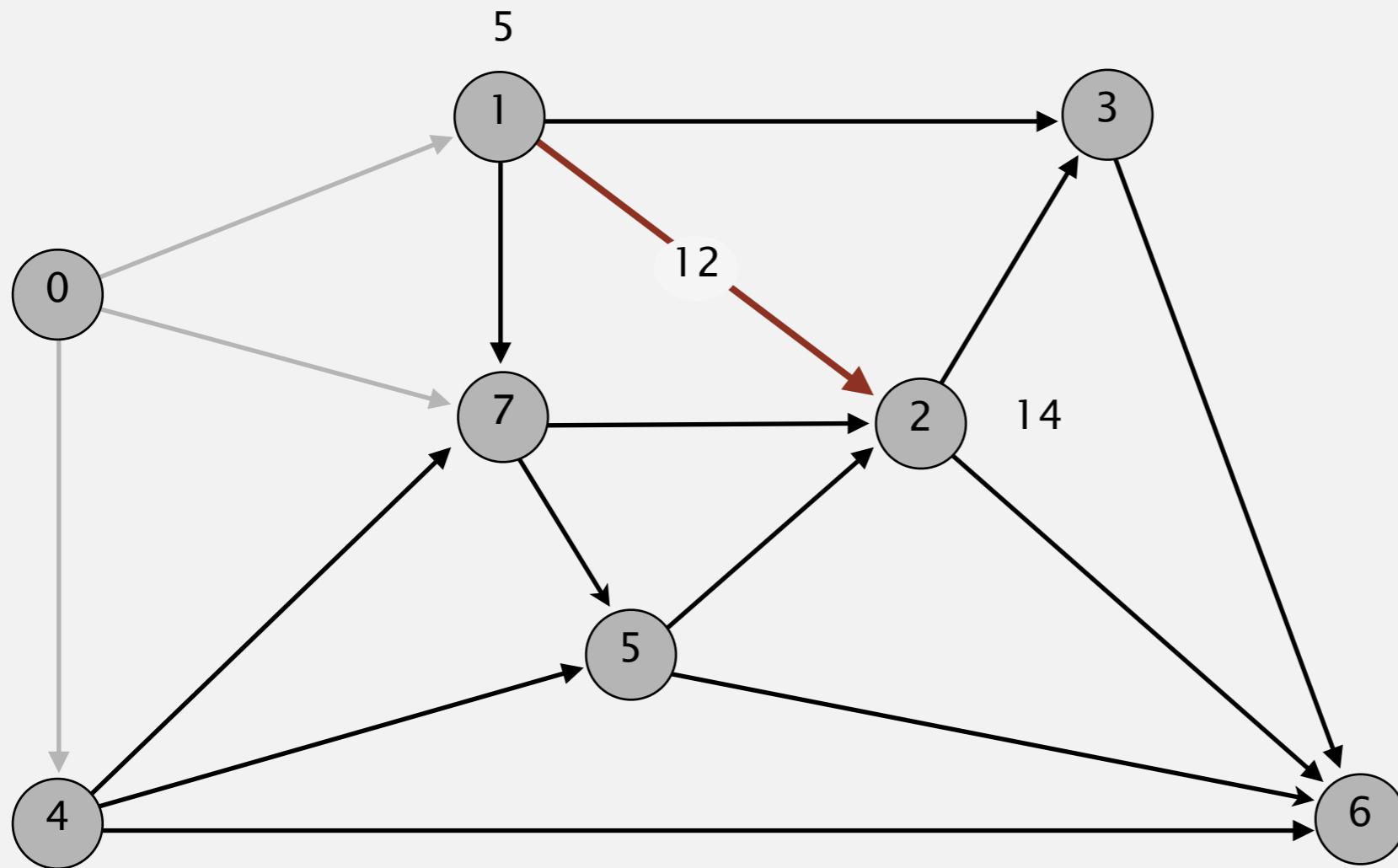
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

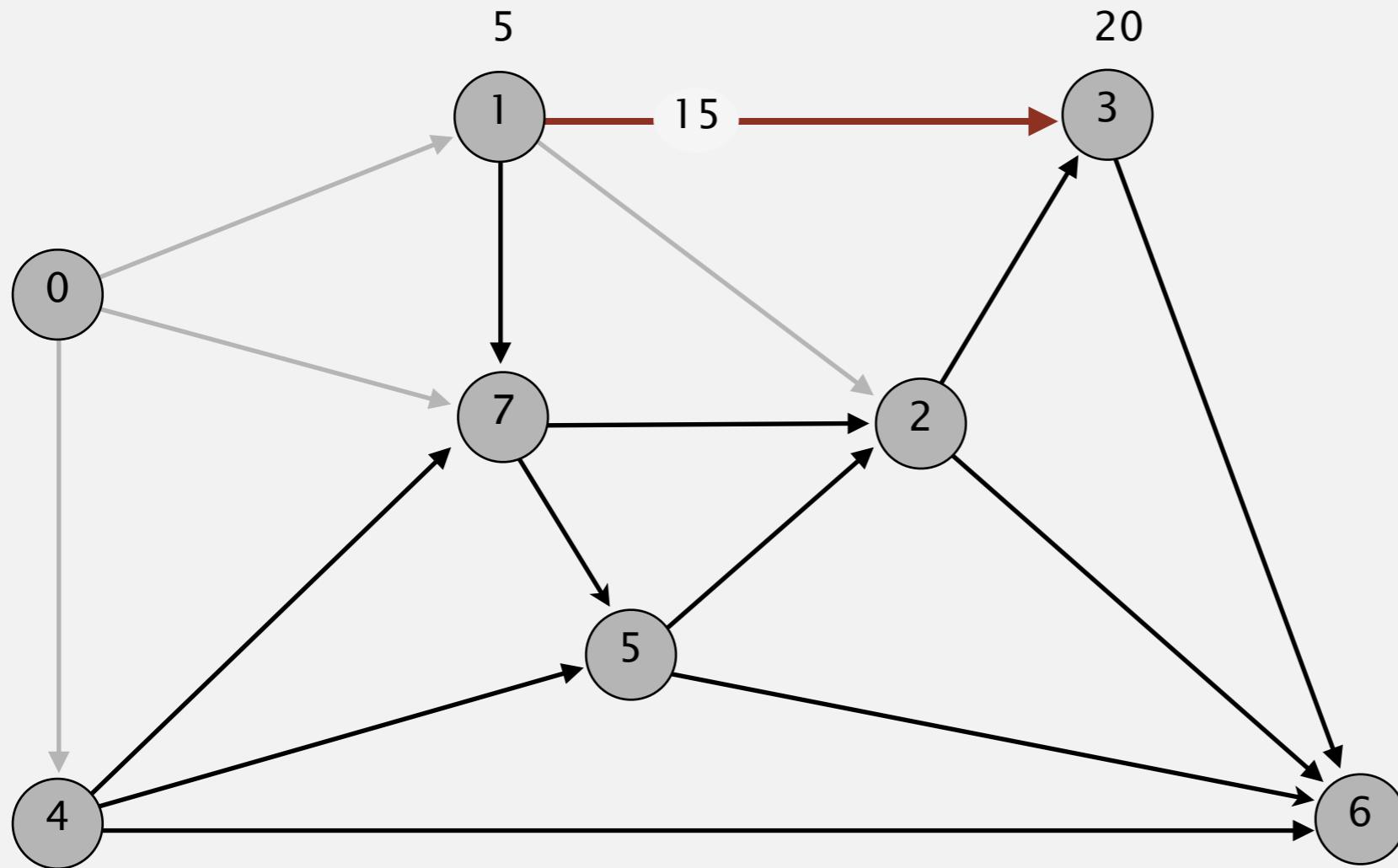
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

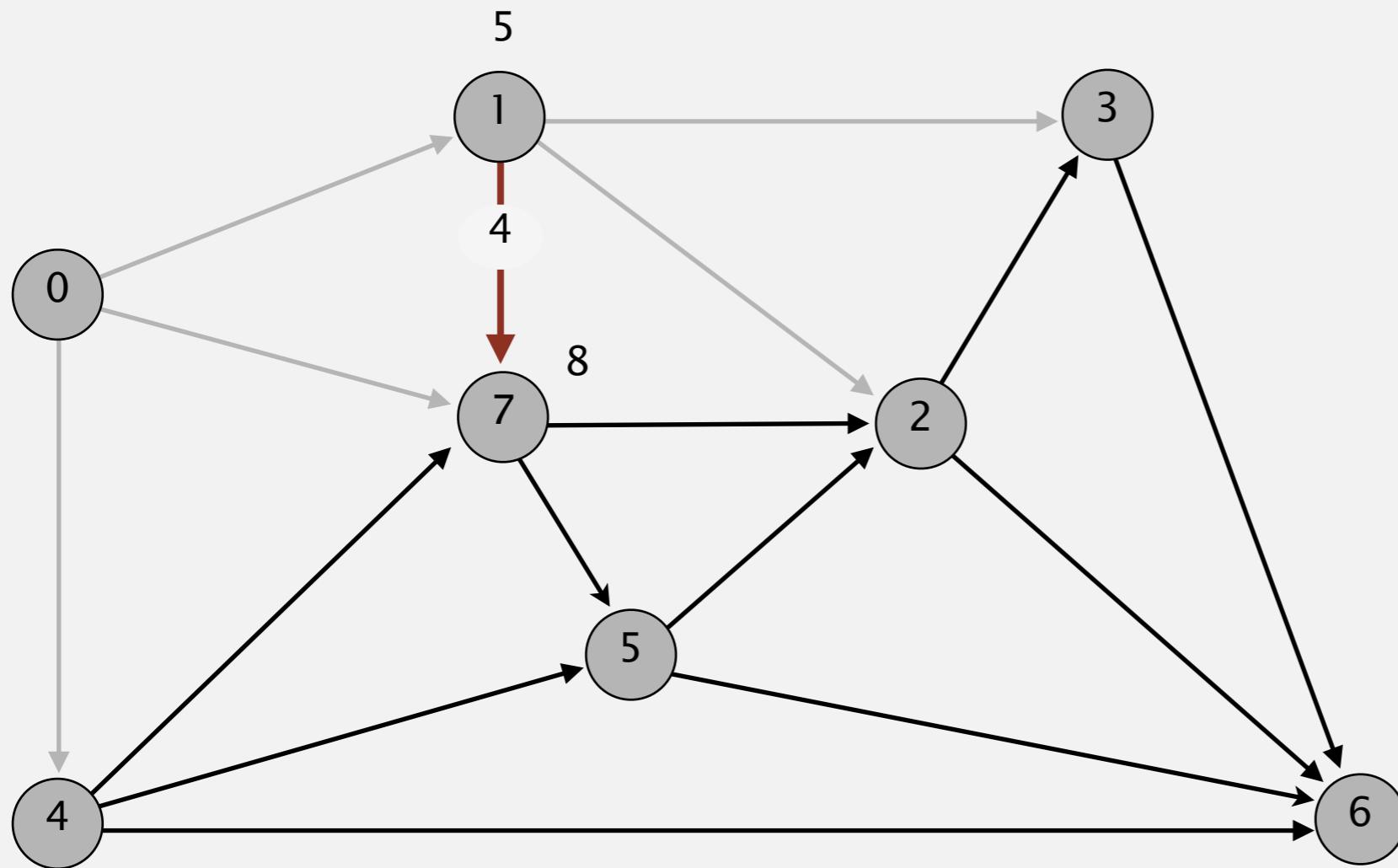
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

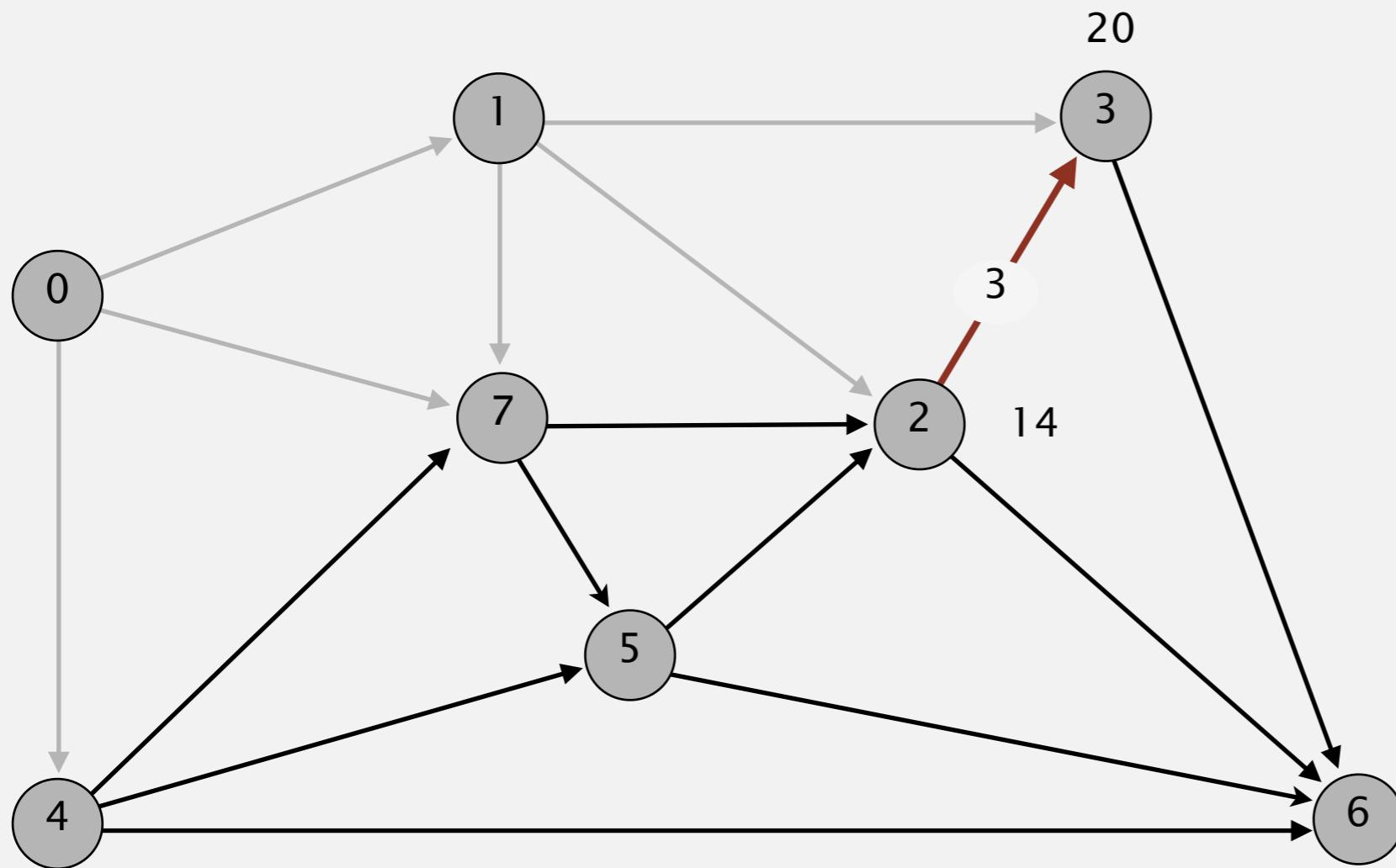
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

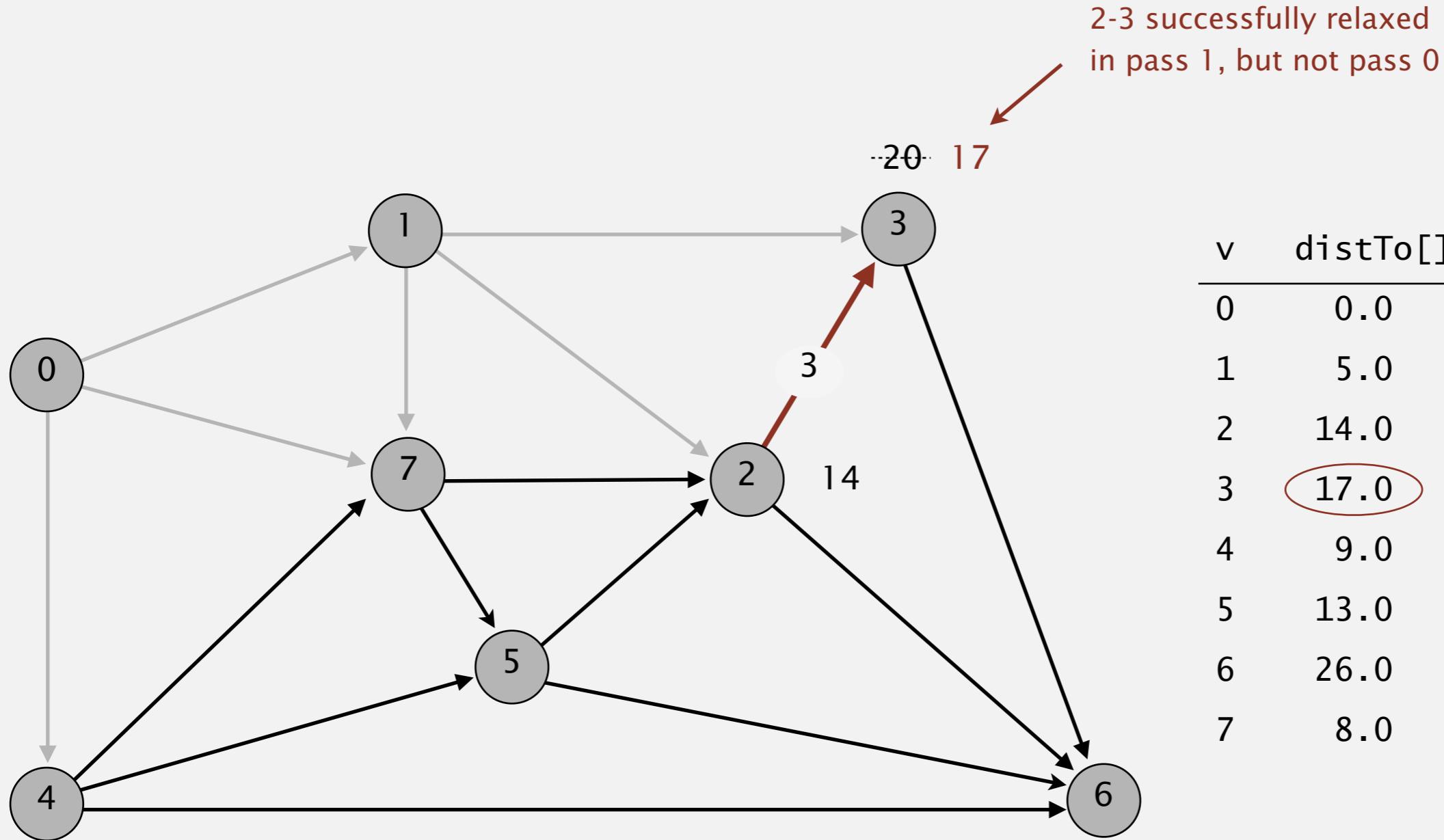
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.

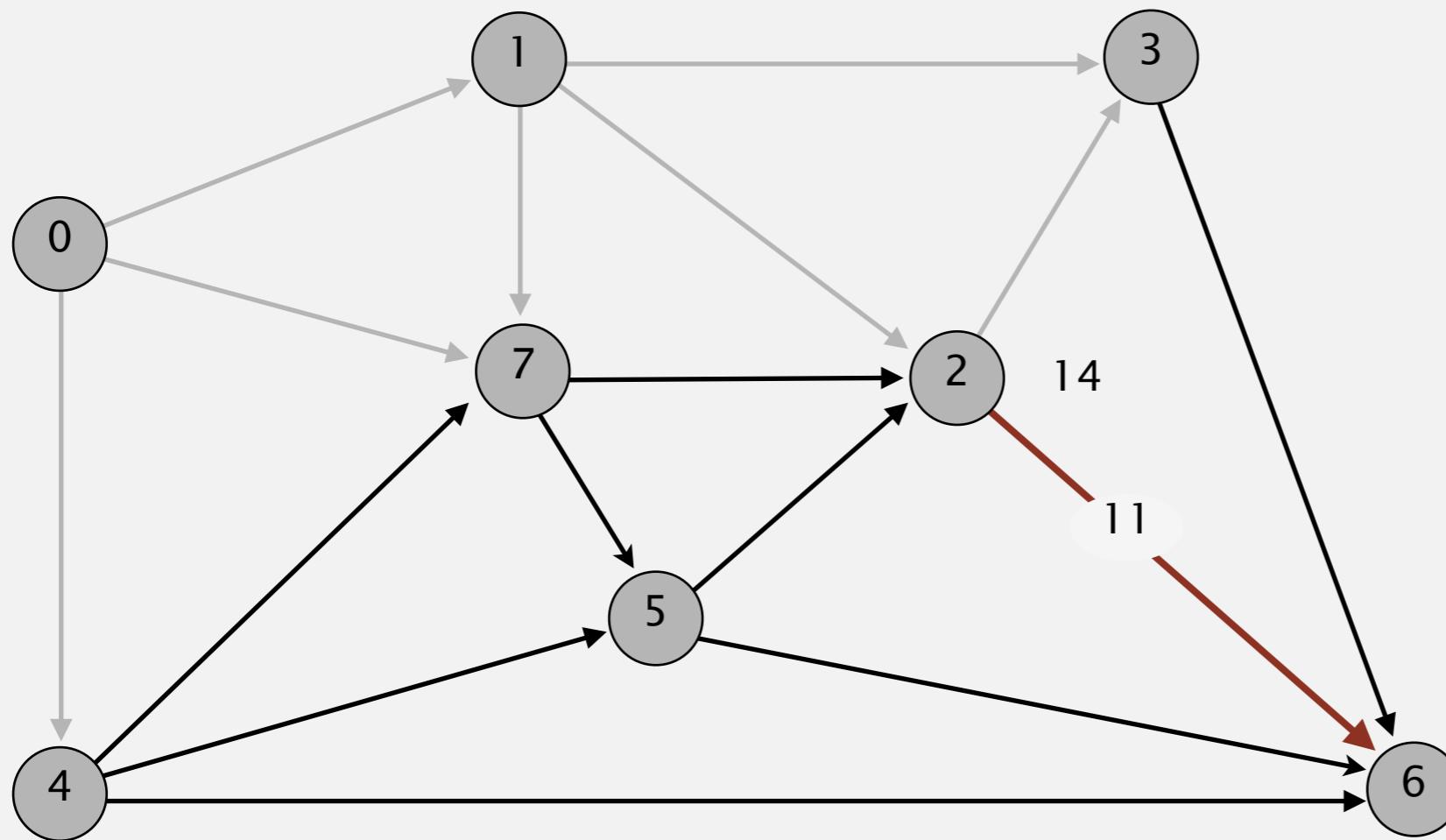


pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



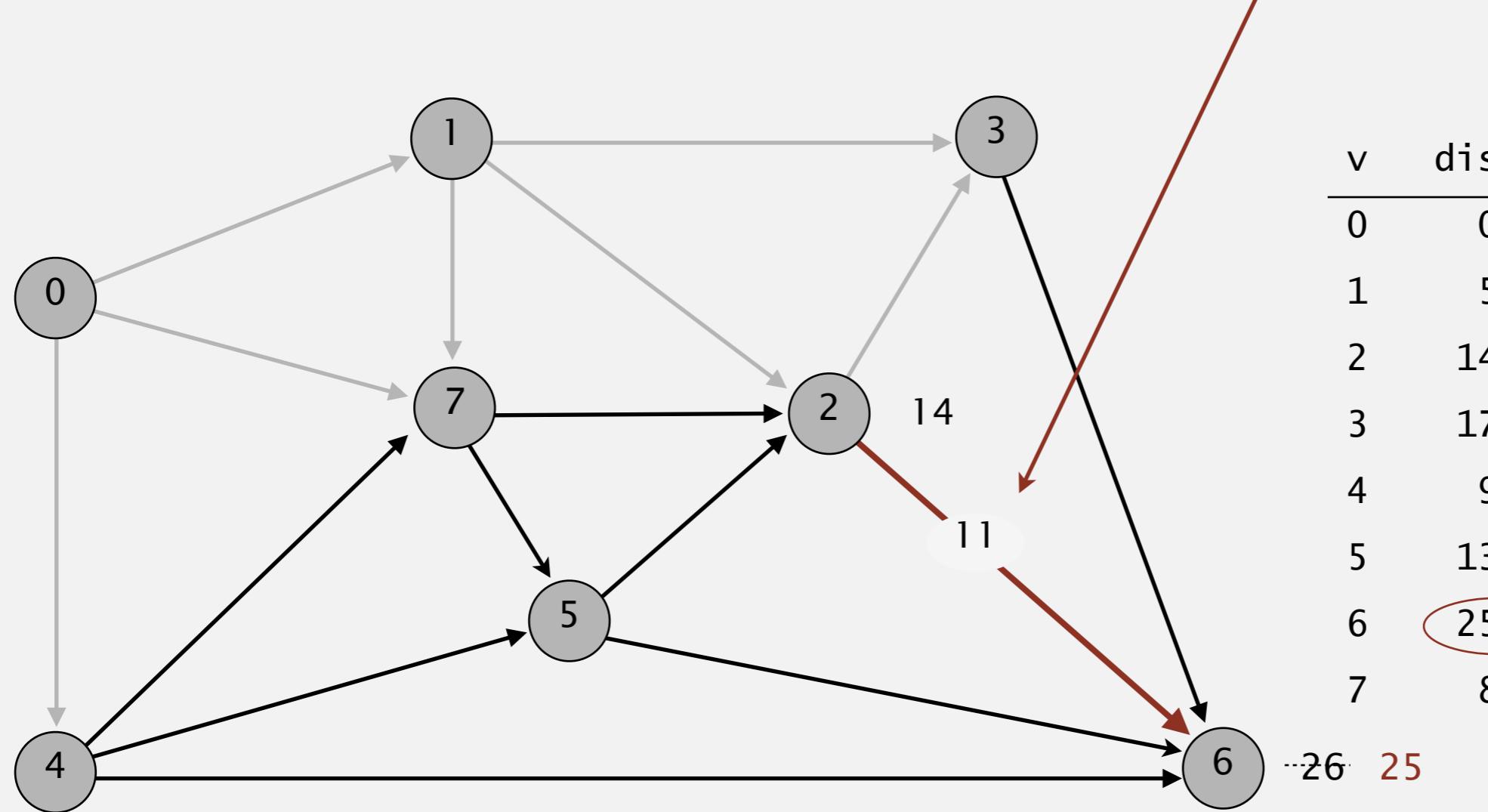
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



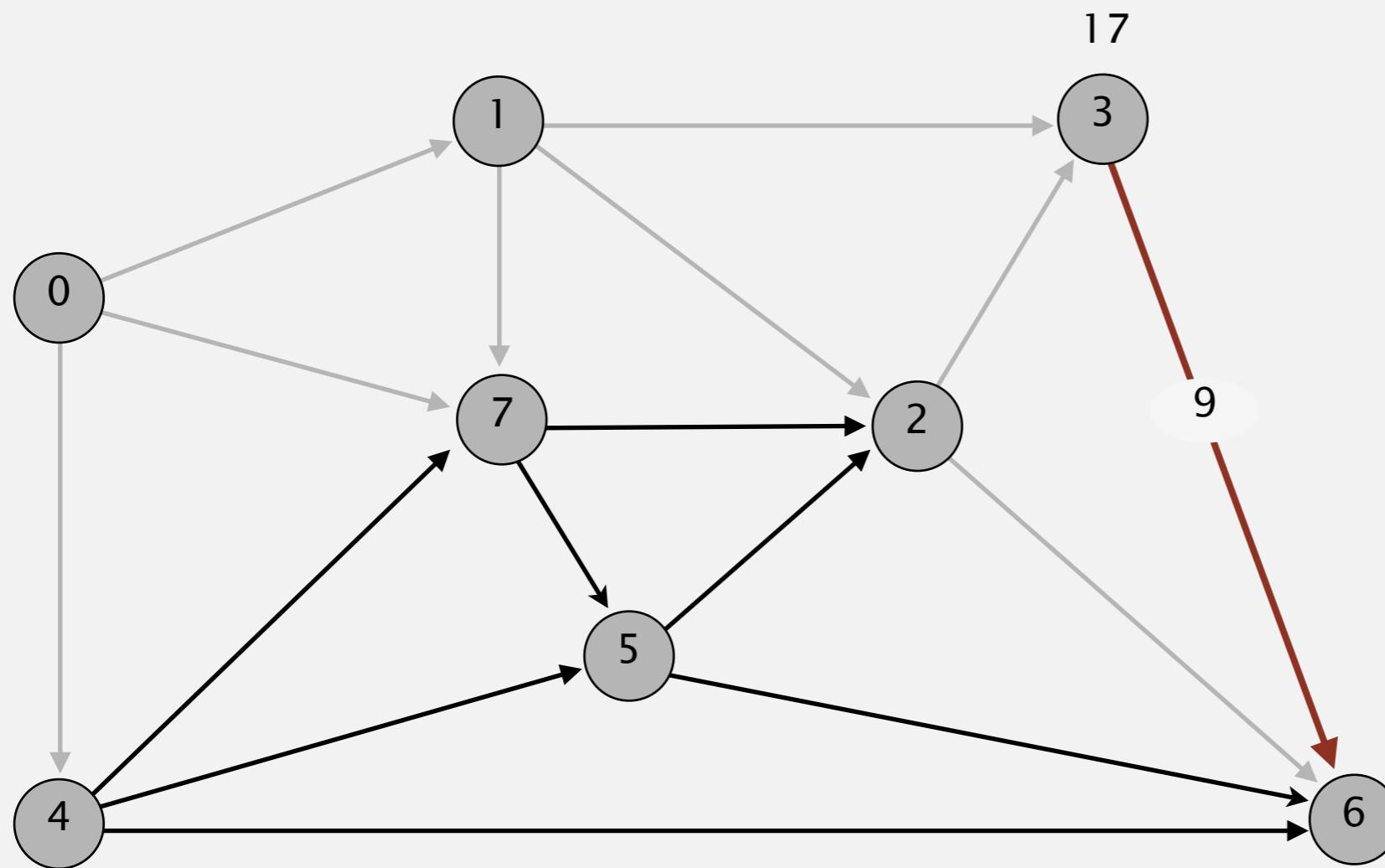
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



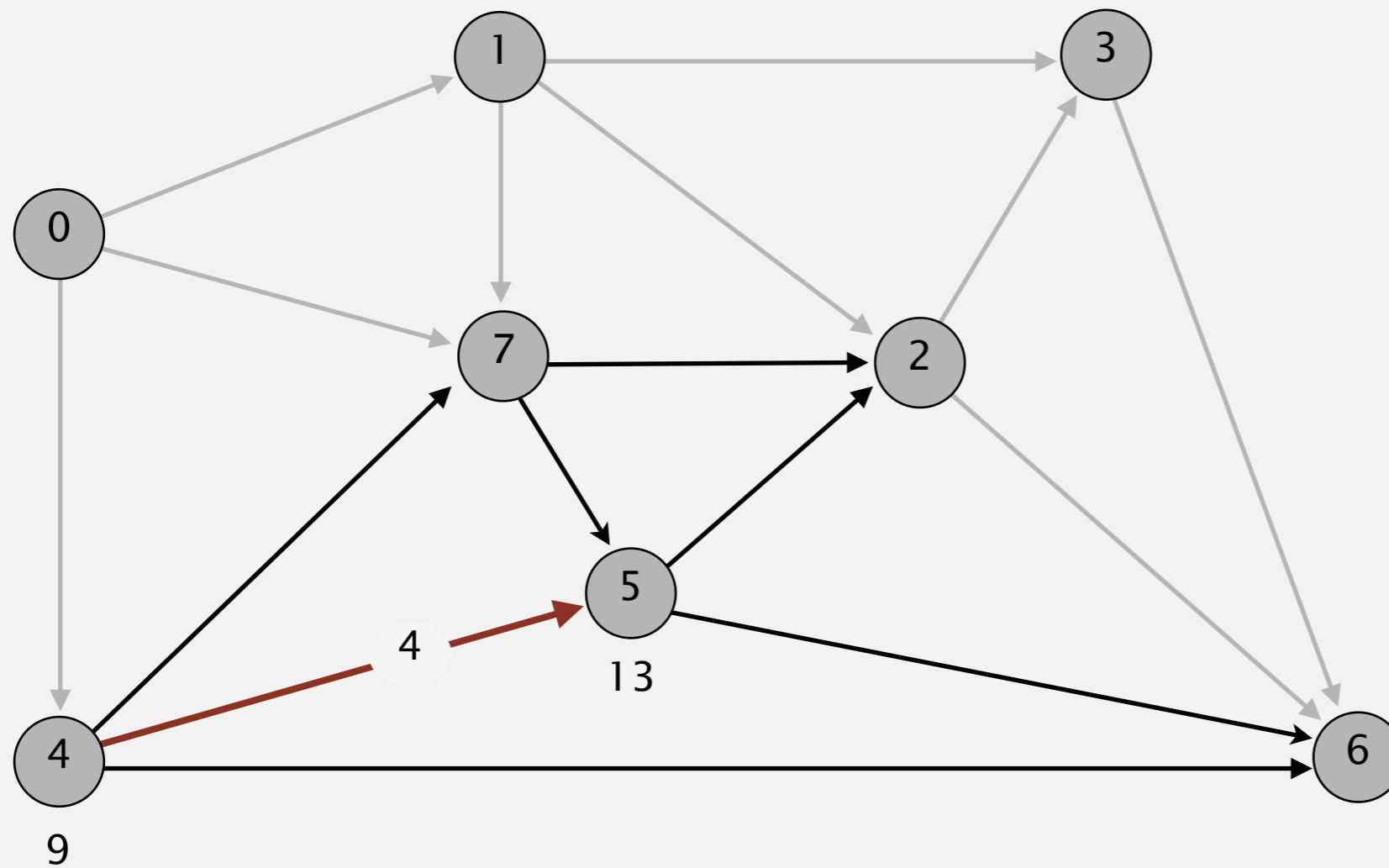
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 6→7 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



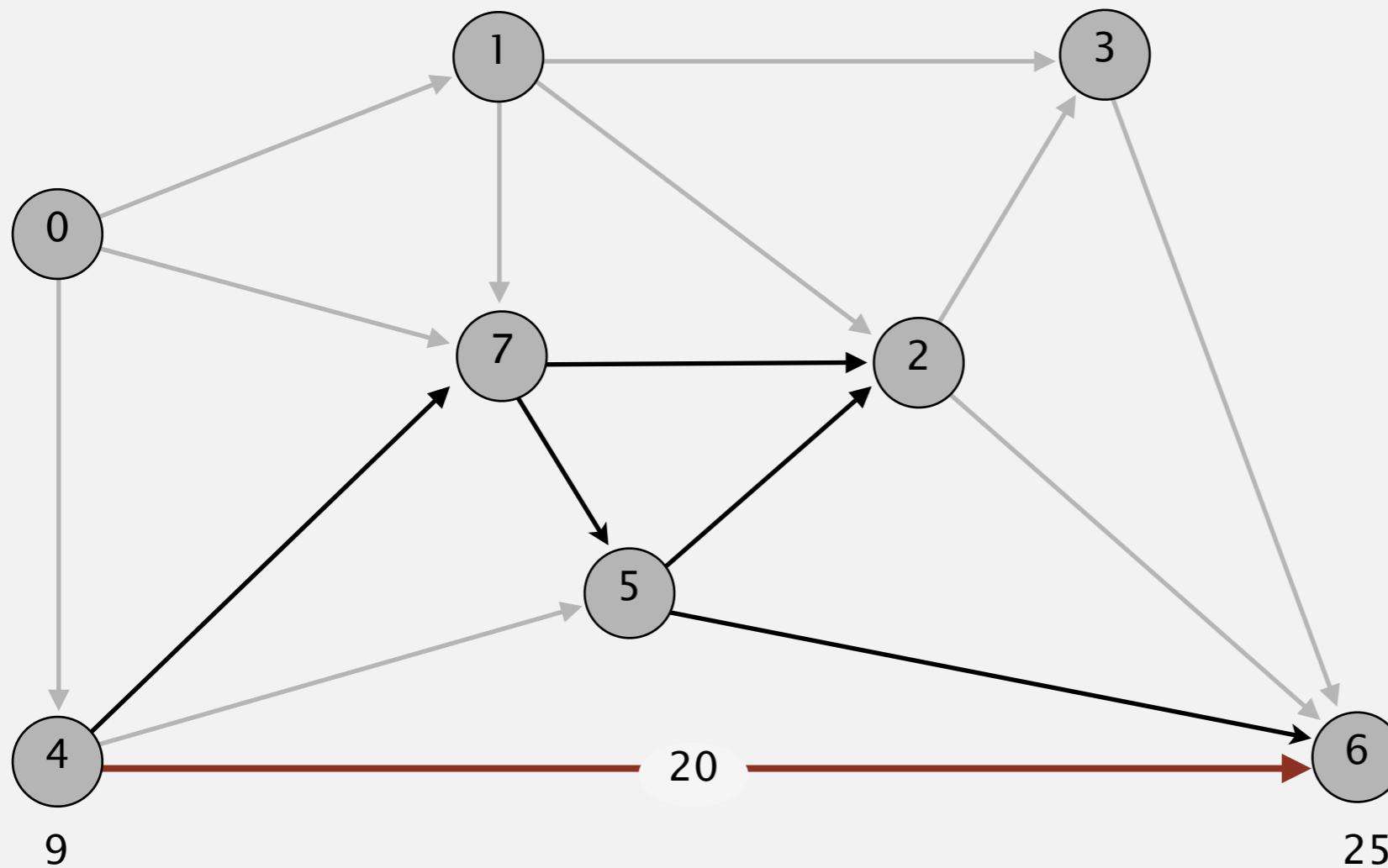
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



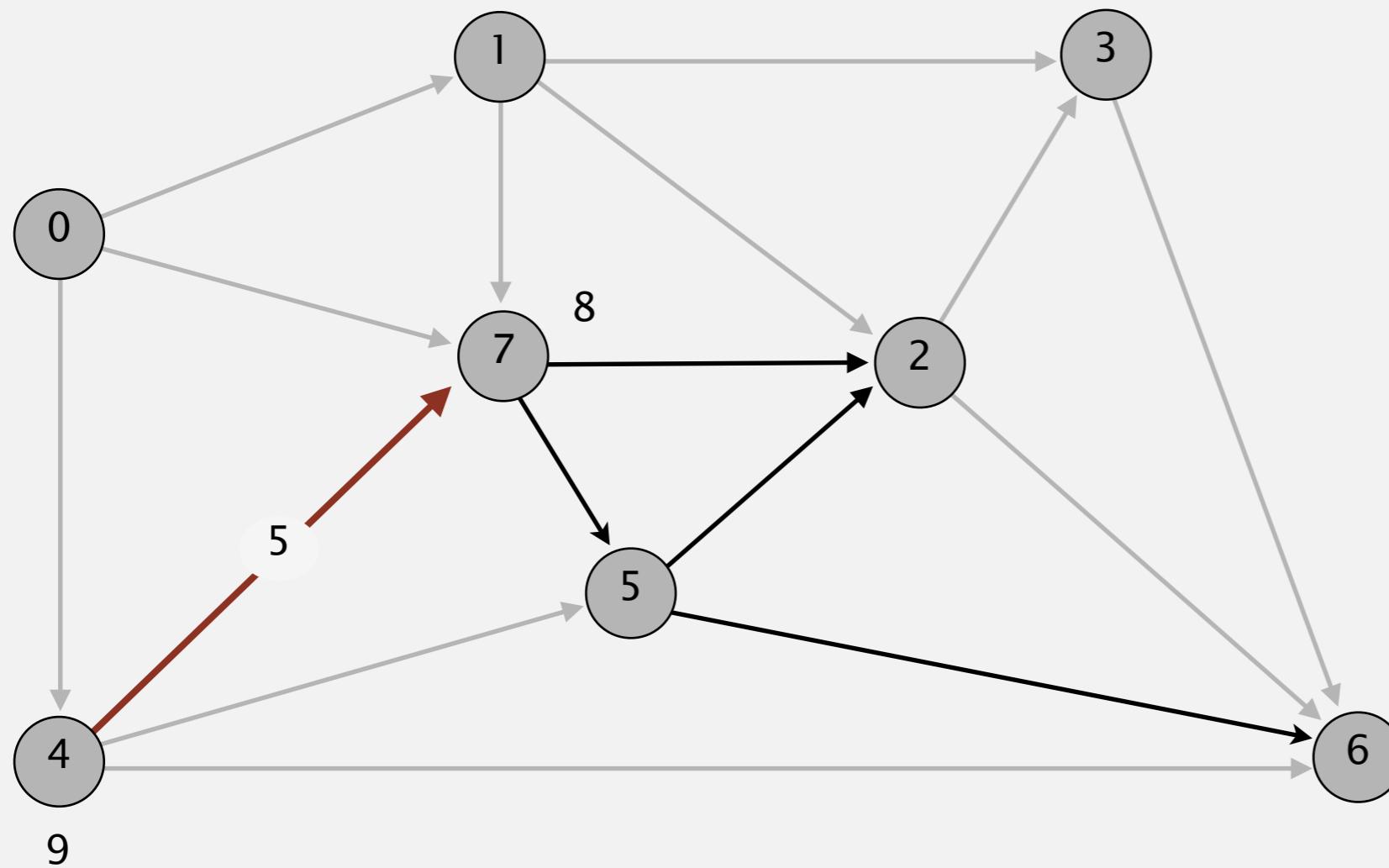
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



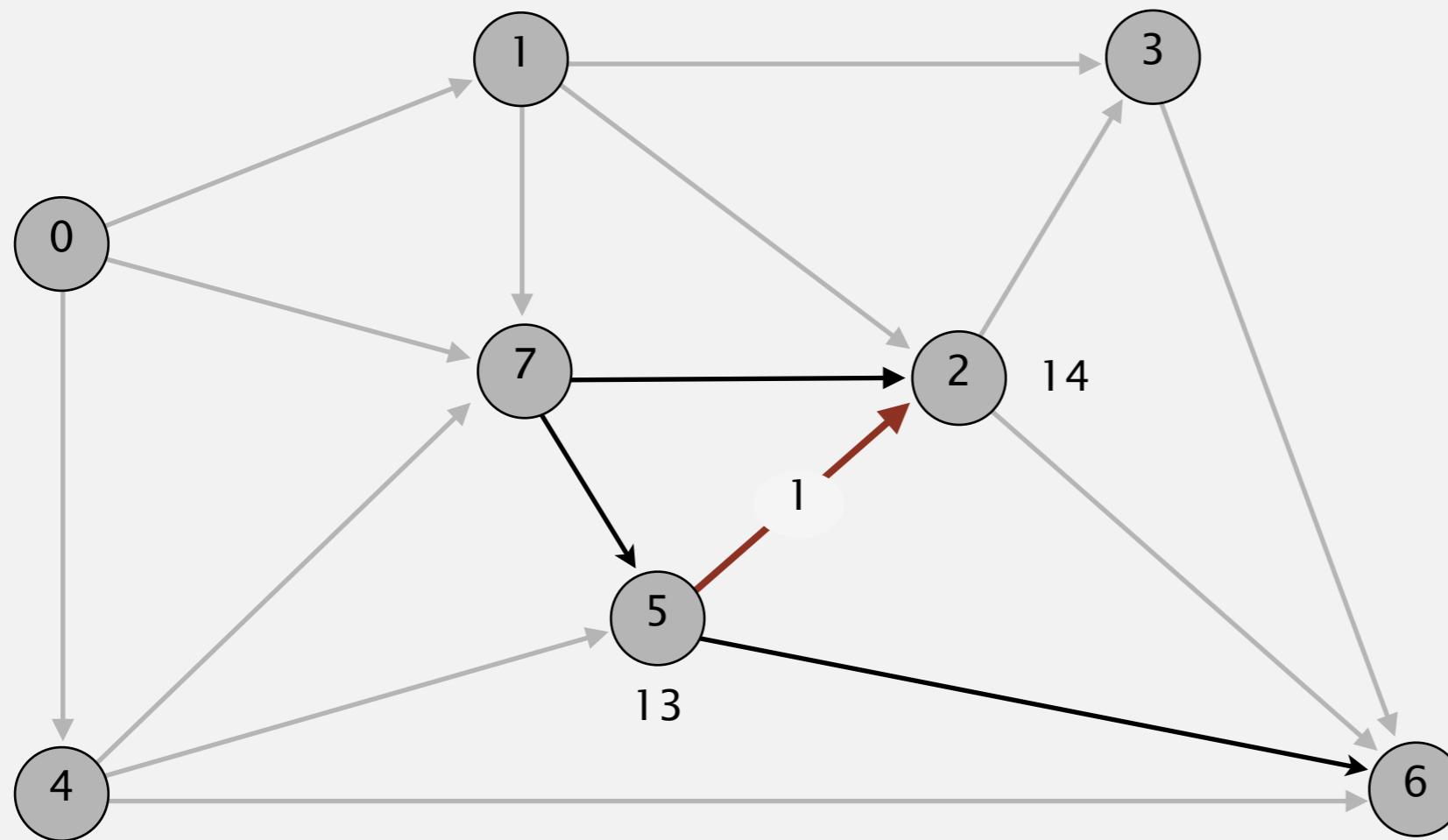
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



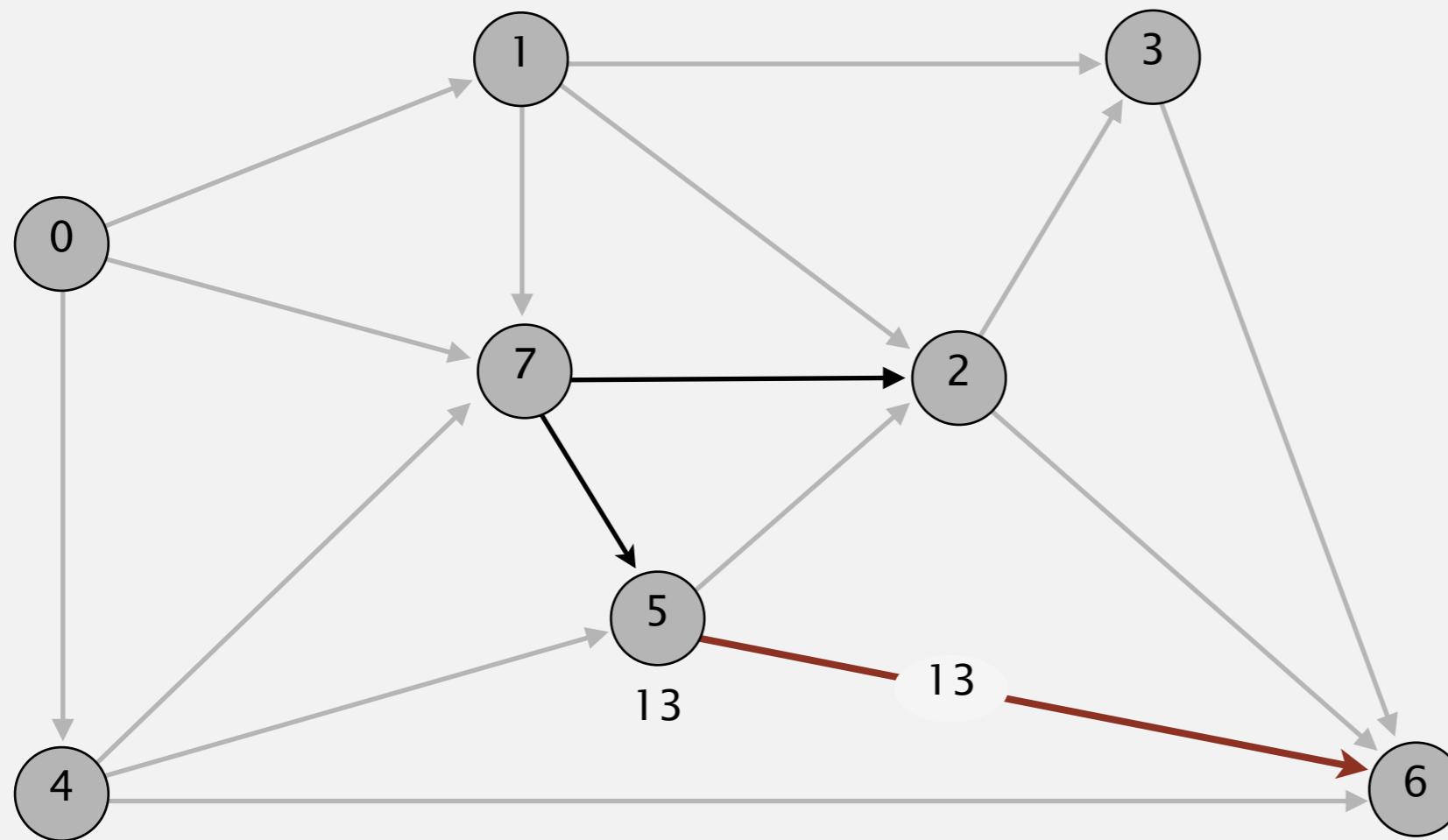
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



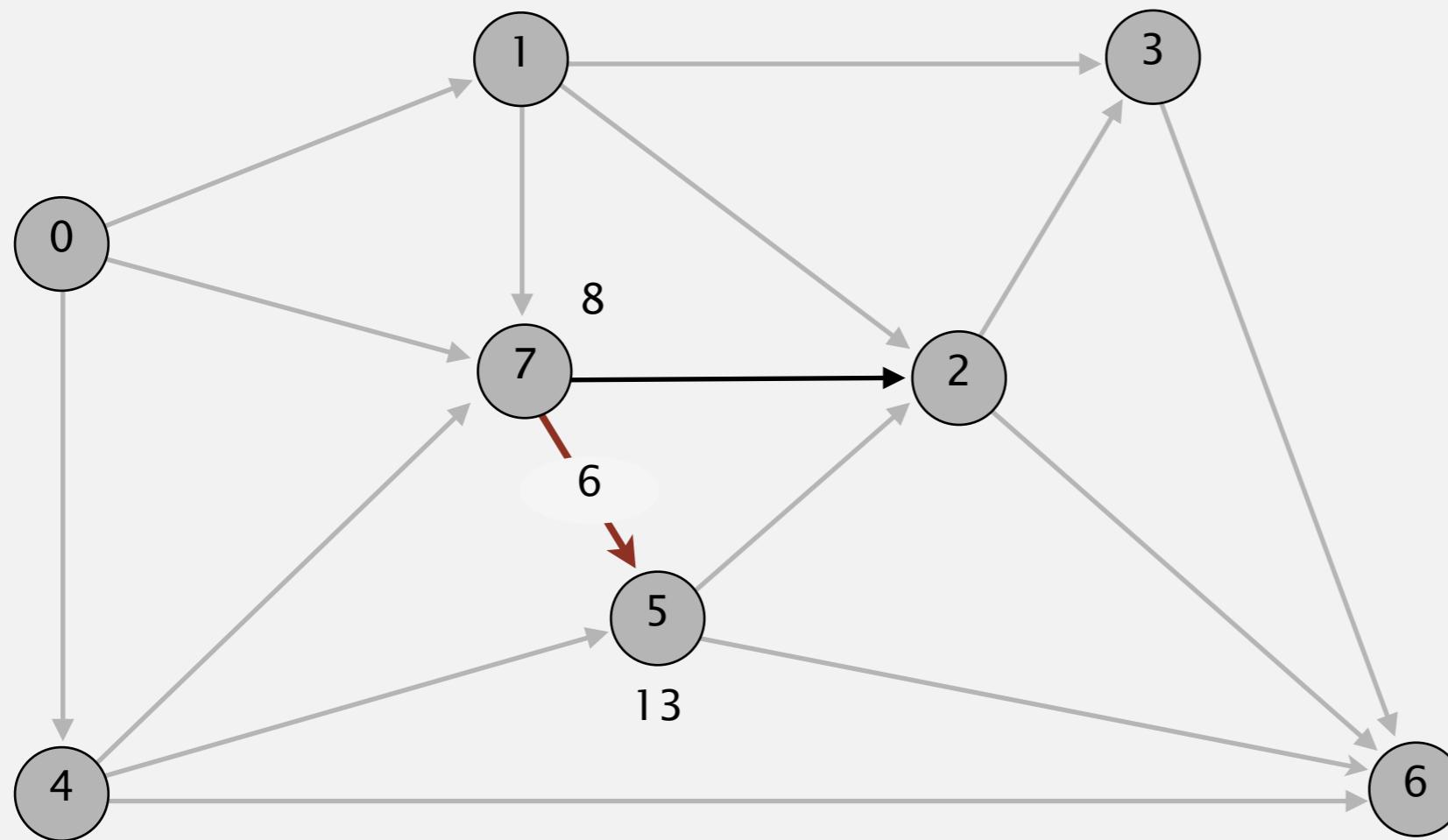
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

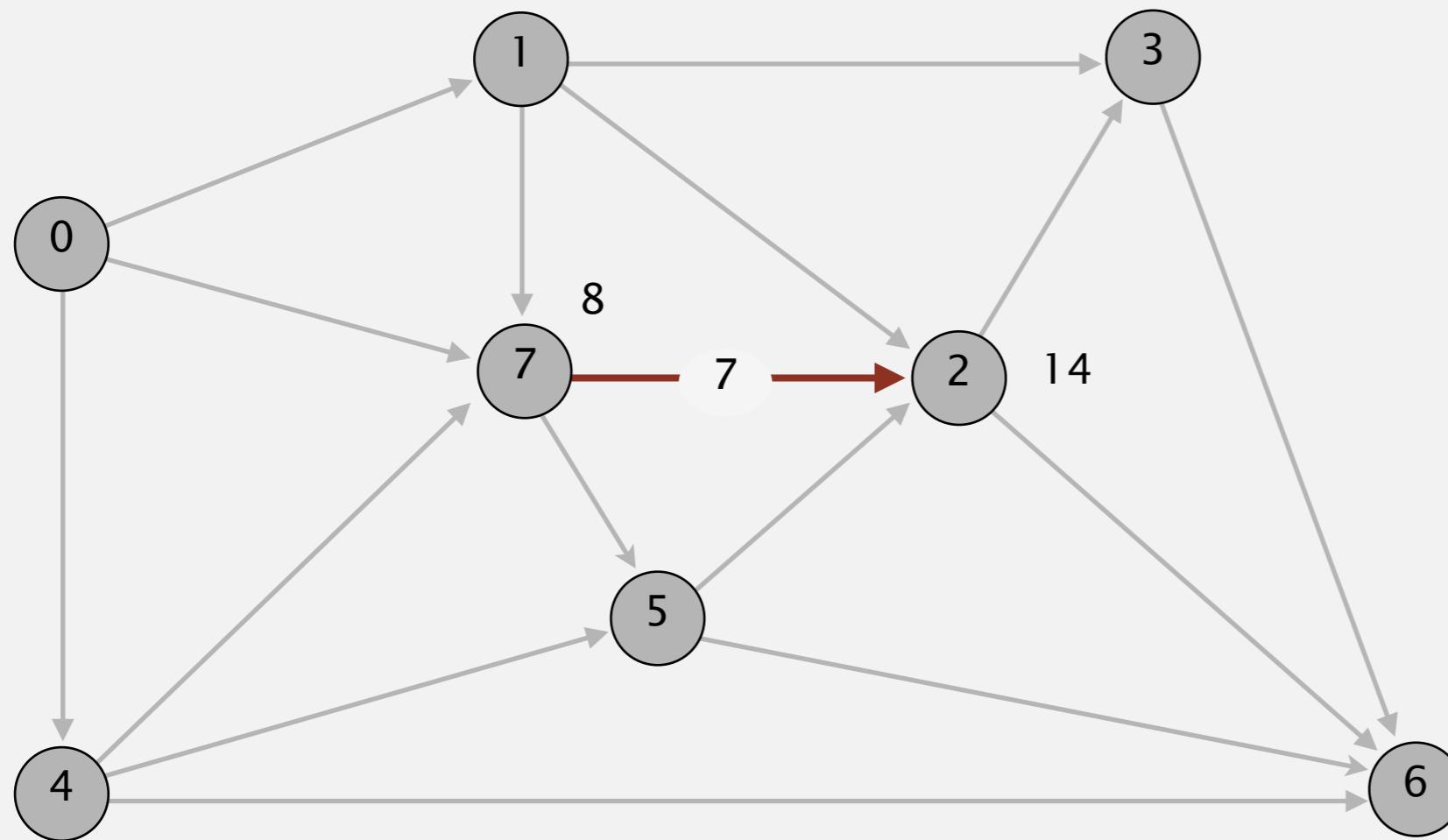
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

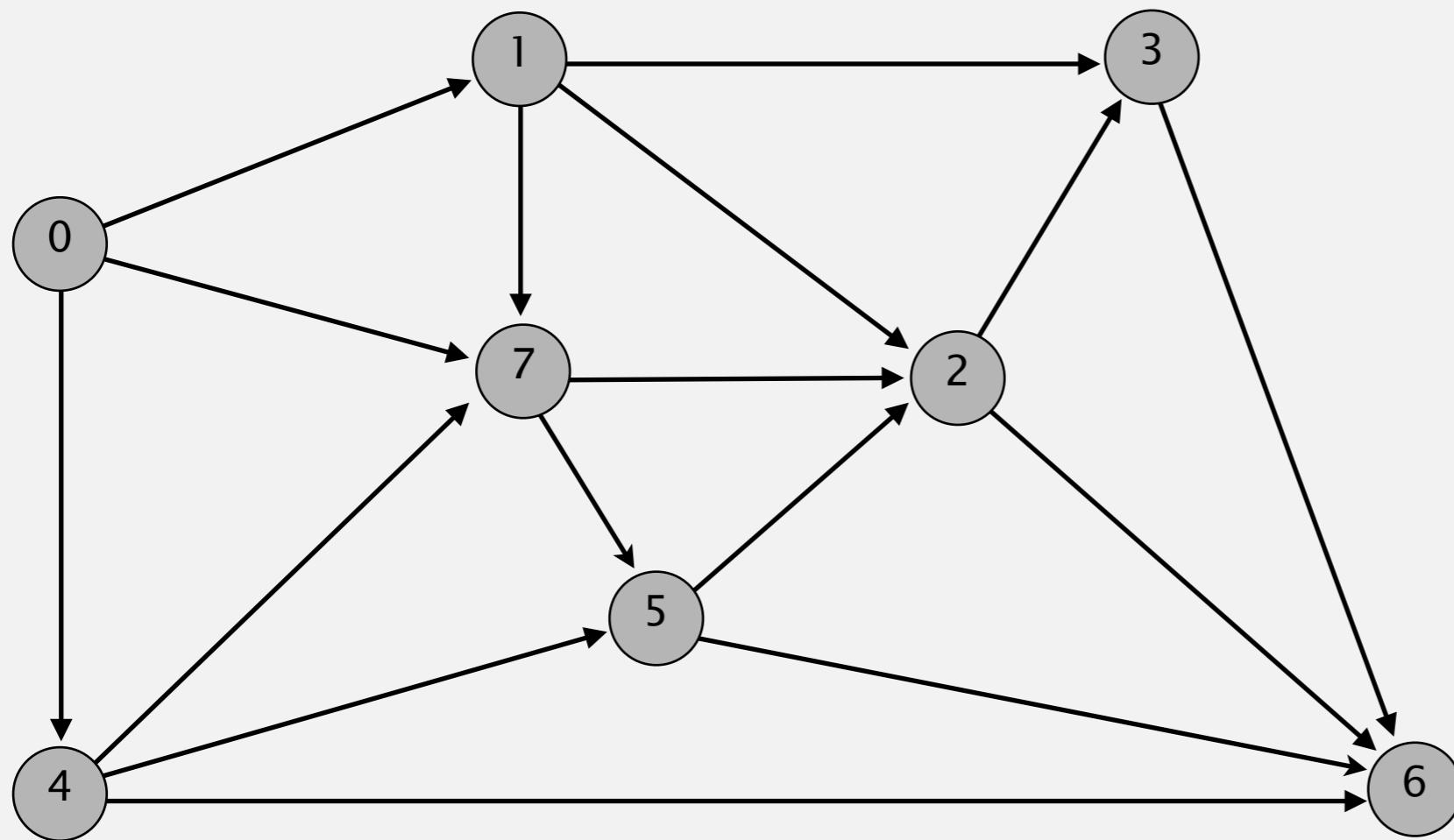
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

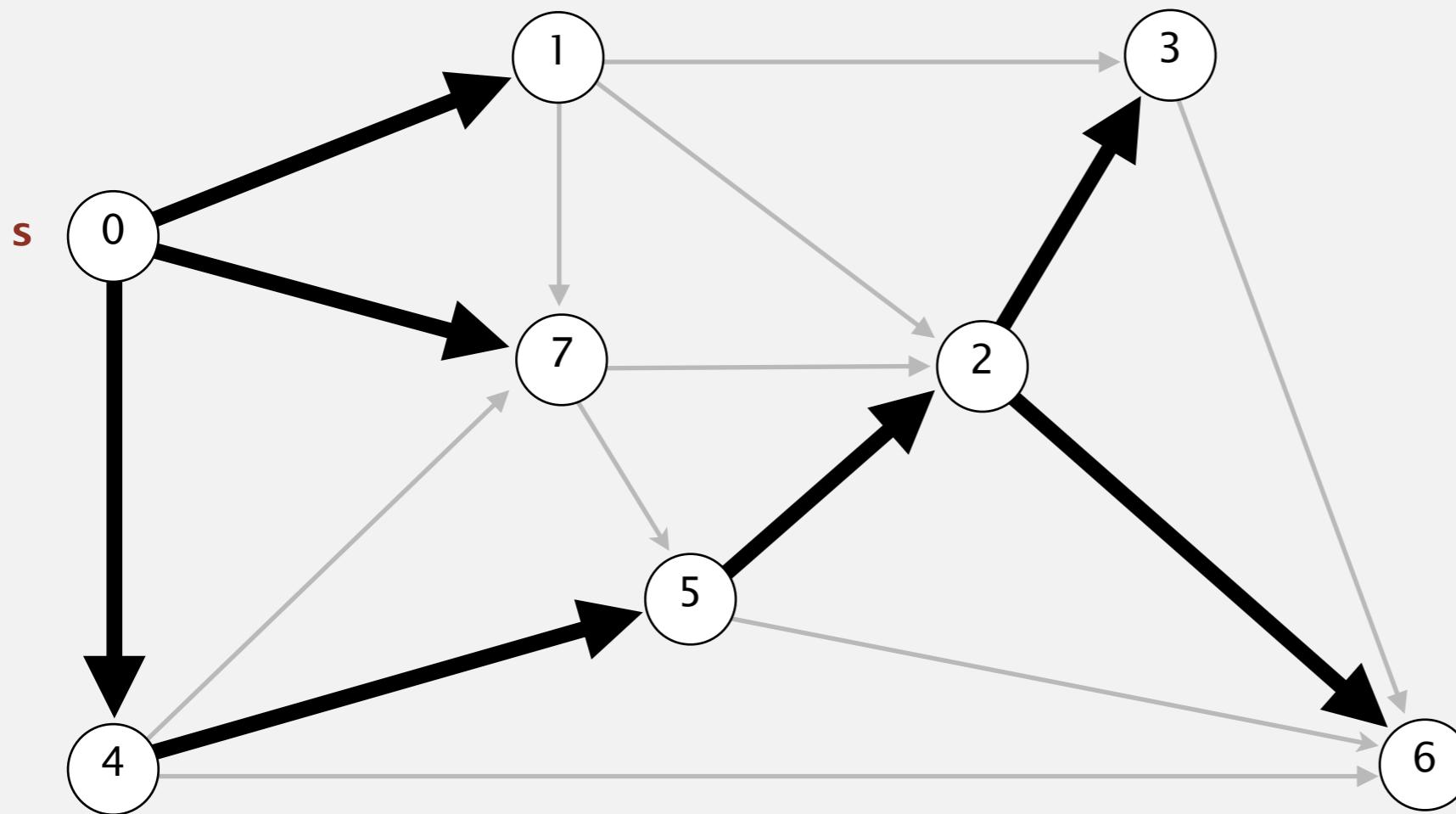
pass 2, 3, 4, 5, 6, 7 (no further changes)

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 6→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

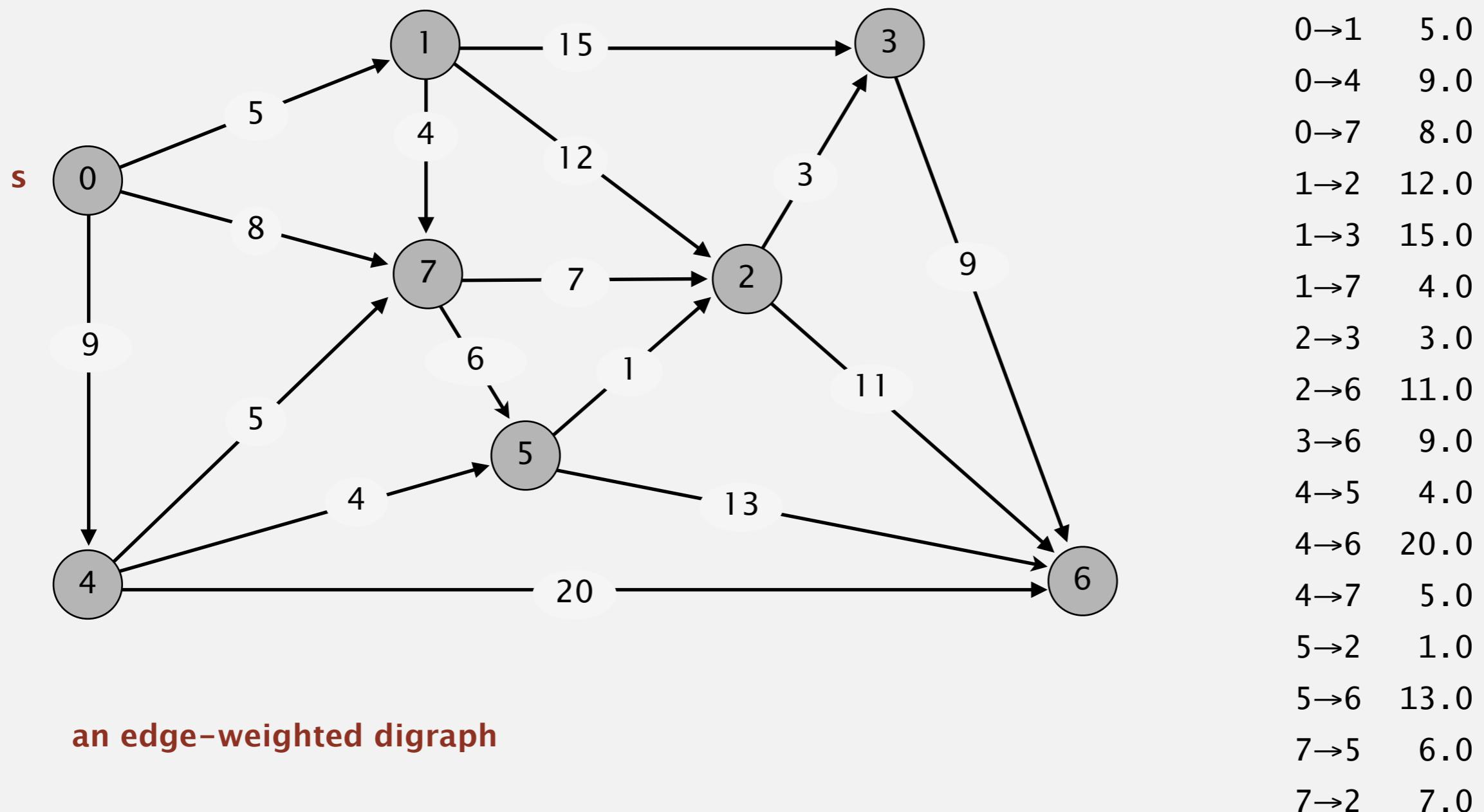


DIJKSTRA'S ALGORITHM DEMO

<http://algs4.cs.princeton.edu>

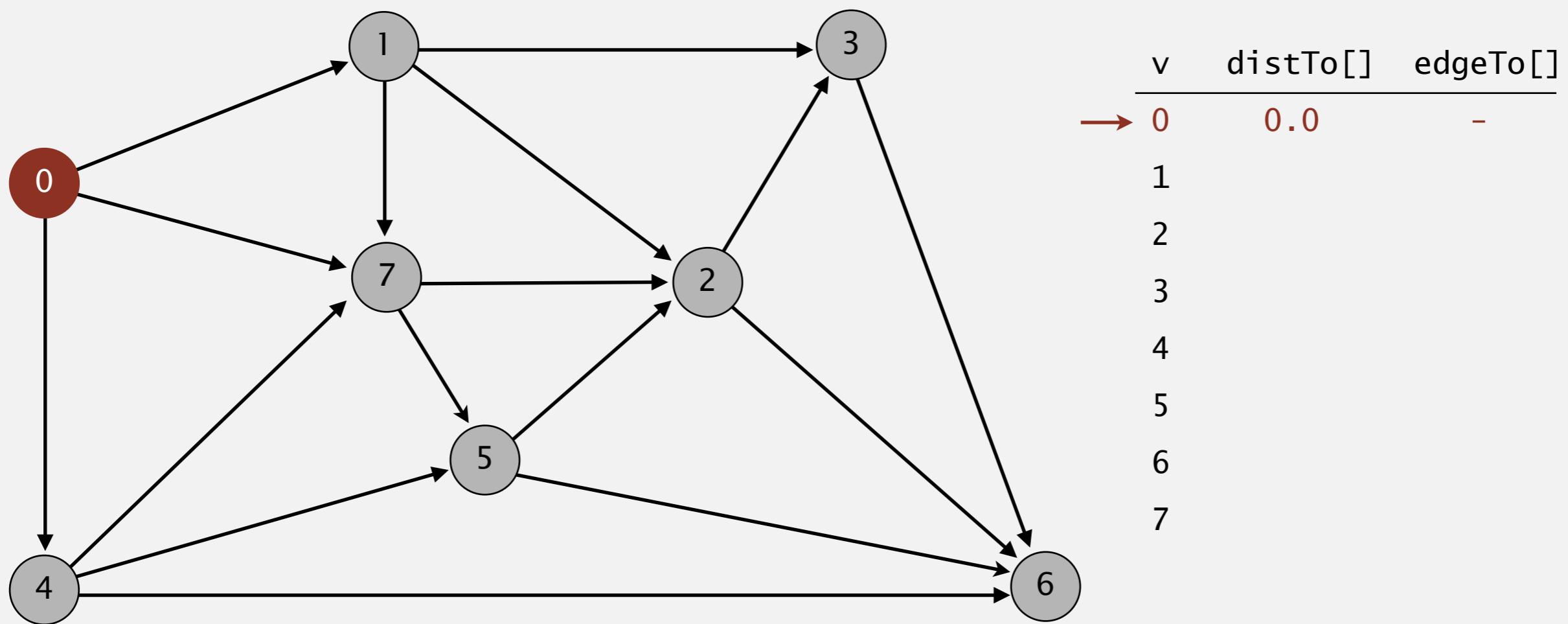
Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



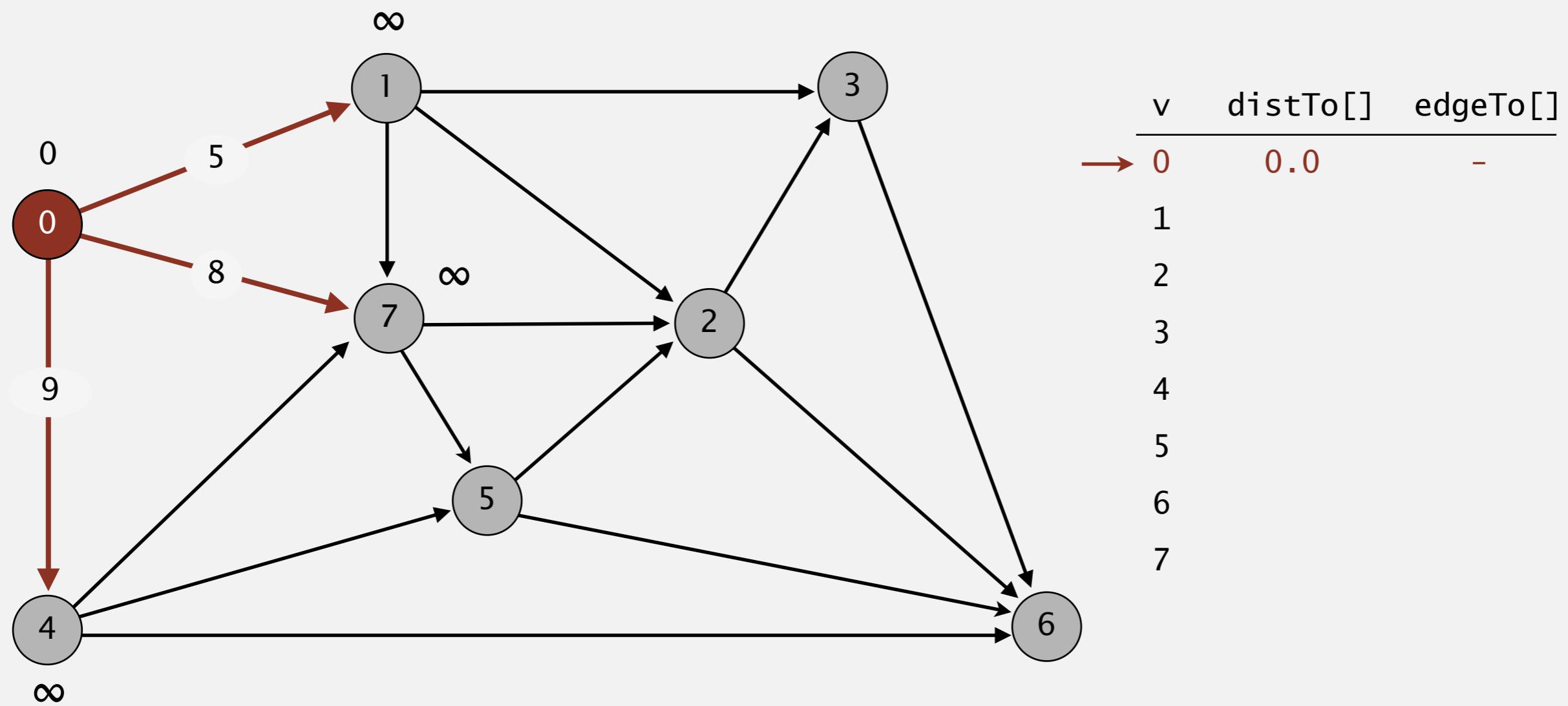
Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



Dijkstra's algorithm demo

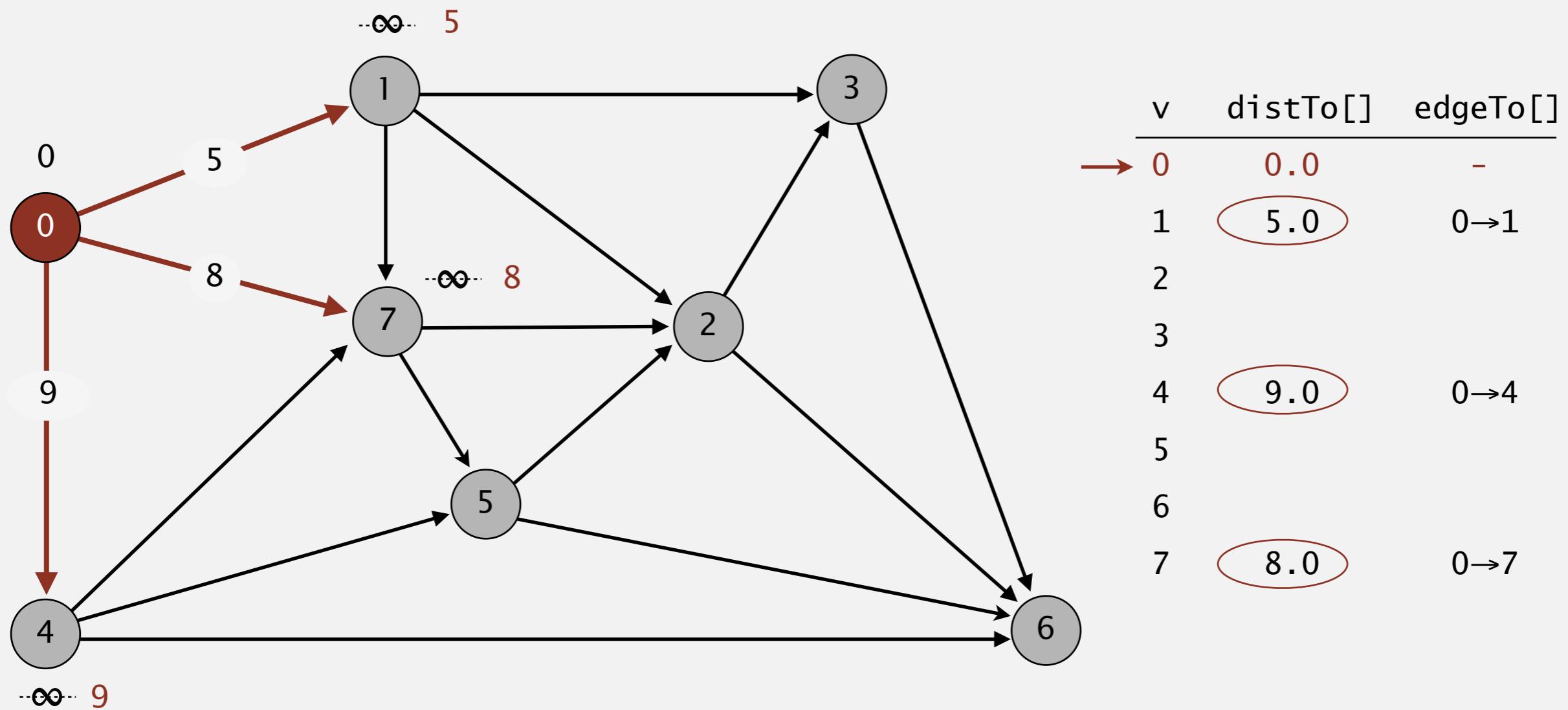
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 0

Dijkstra's algorithm demo

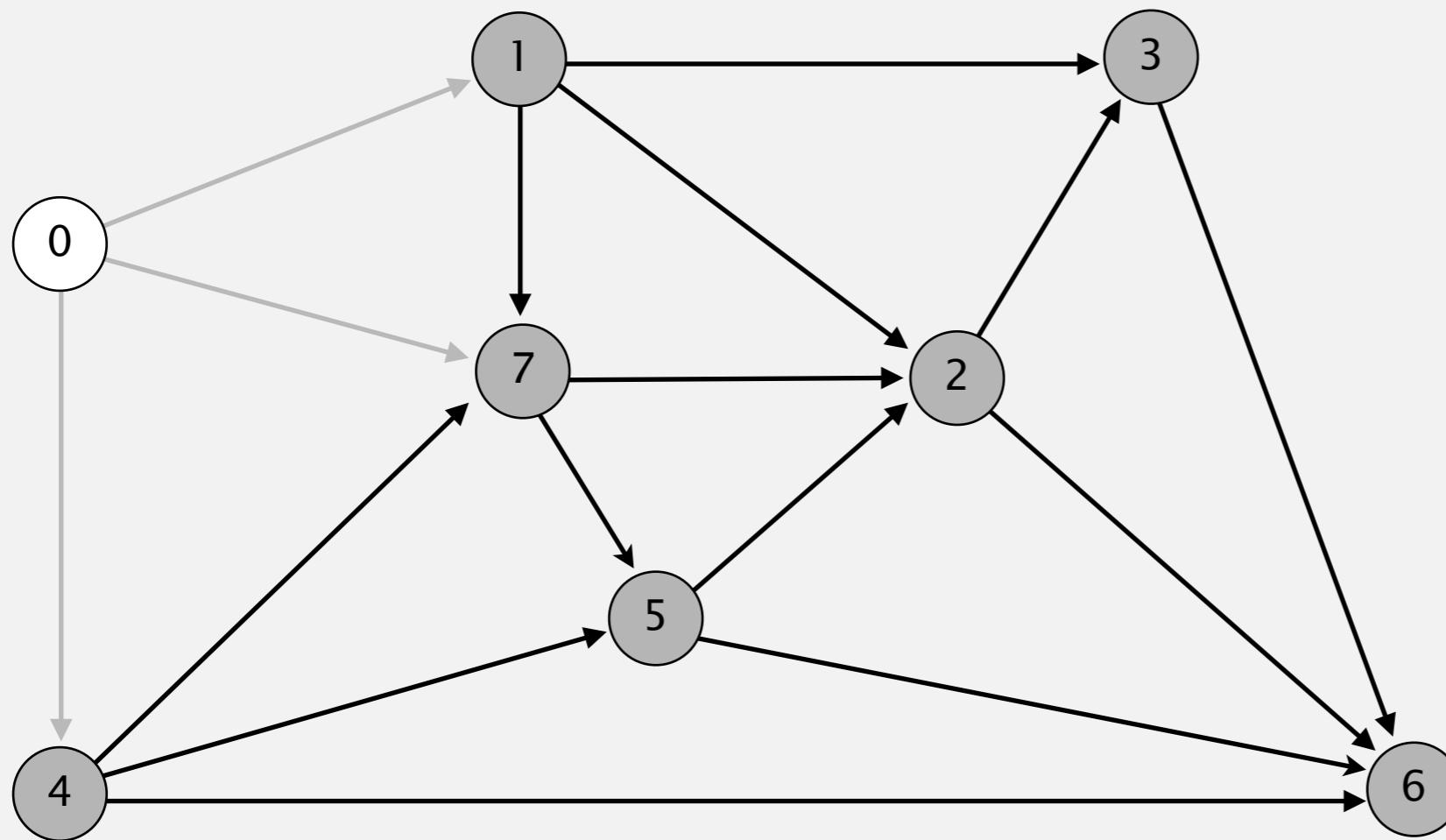
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 0

Dijkstra's algorithm demo

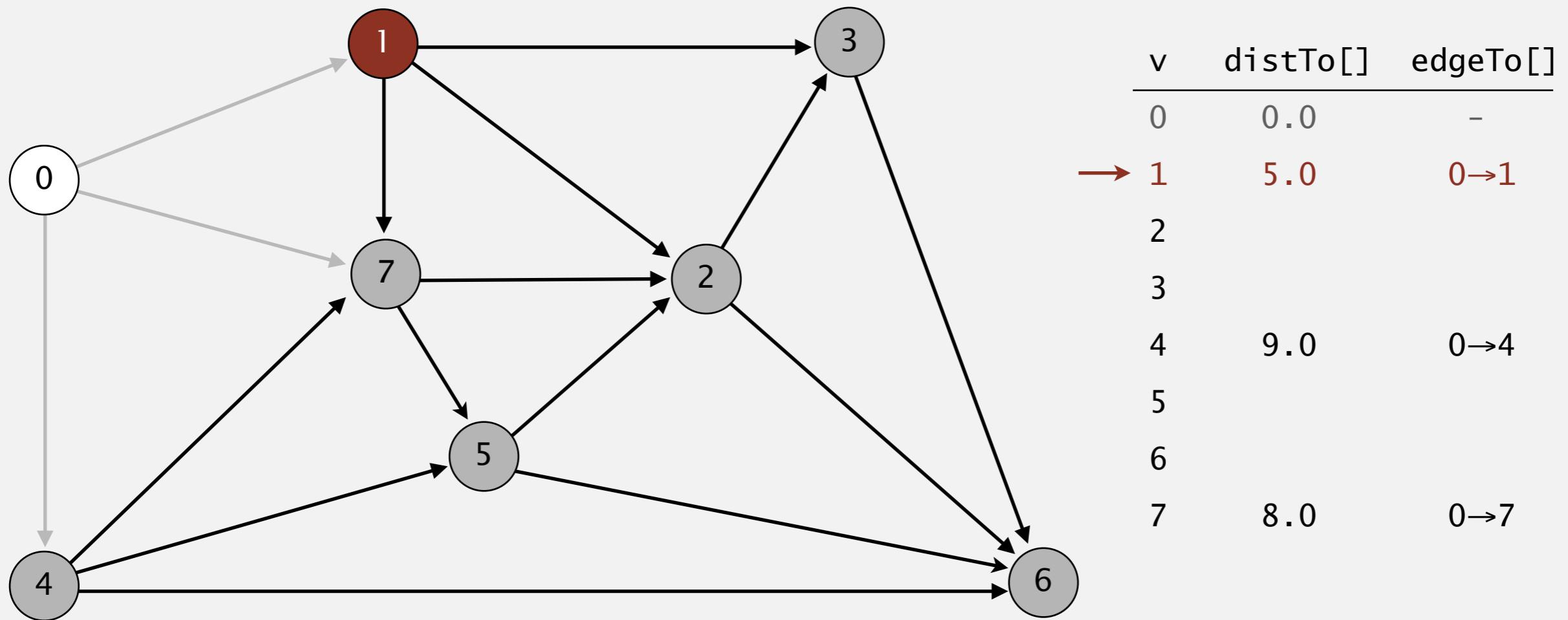
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Dijkstra's algorithm demo

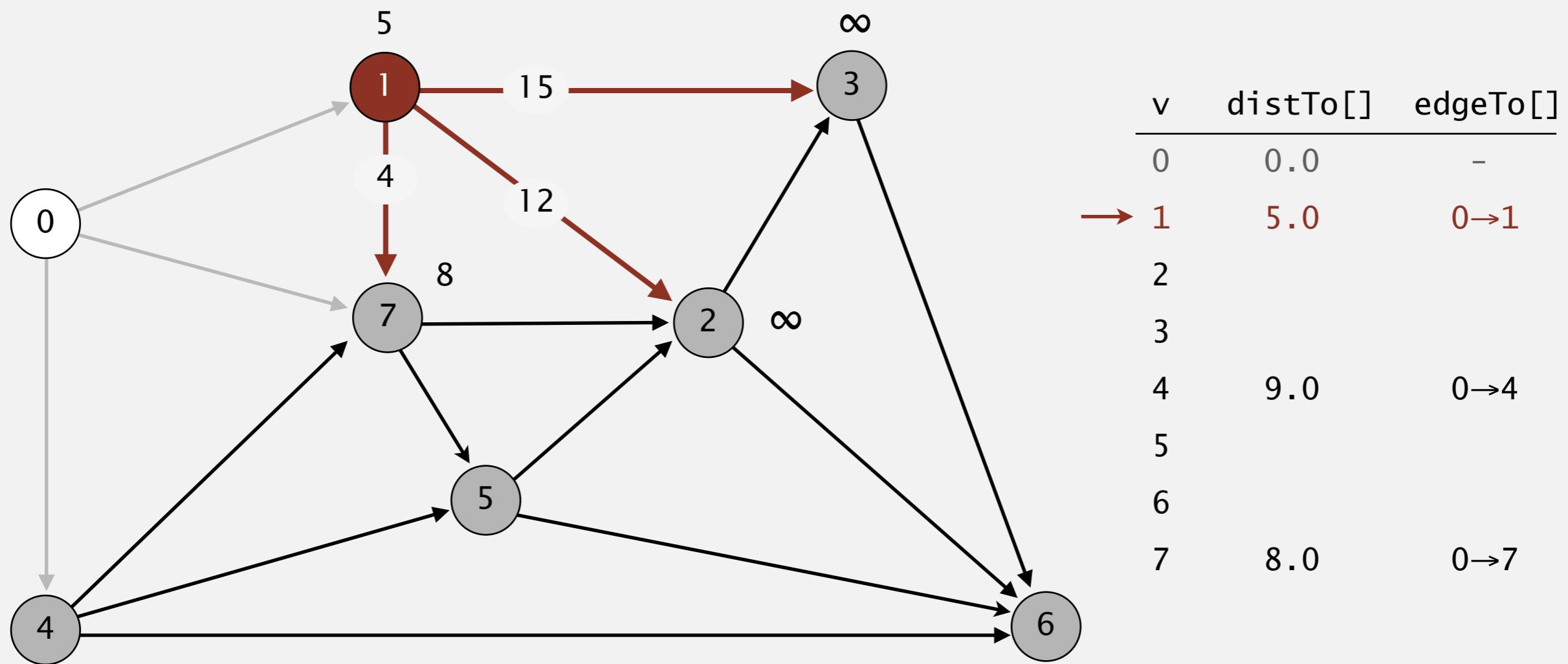
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



choose vertex 1

Dijkstra's algorithm demo

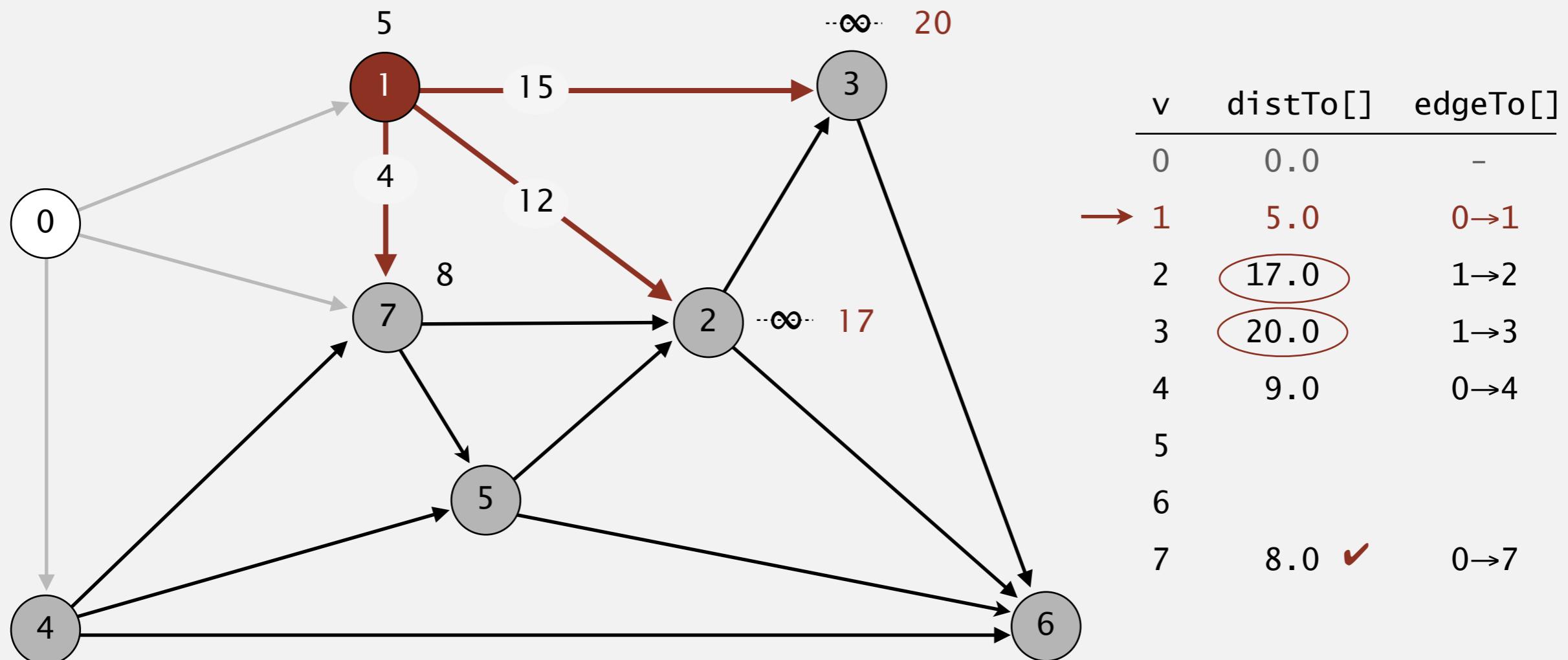
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 1

Dijkstra's algorithm demo

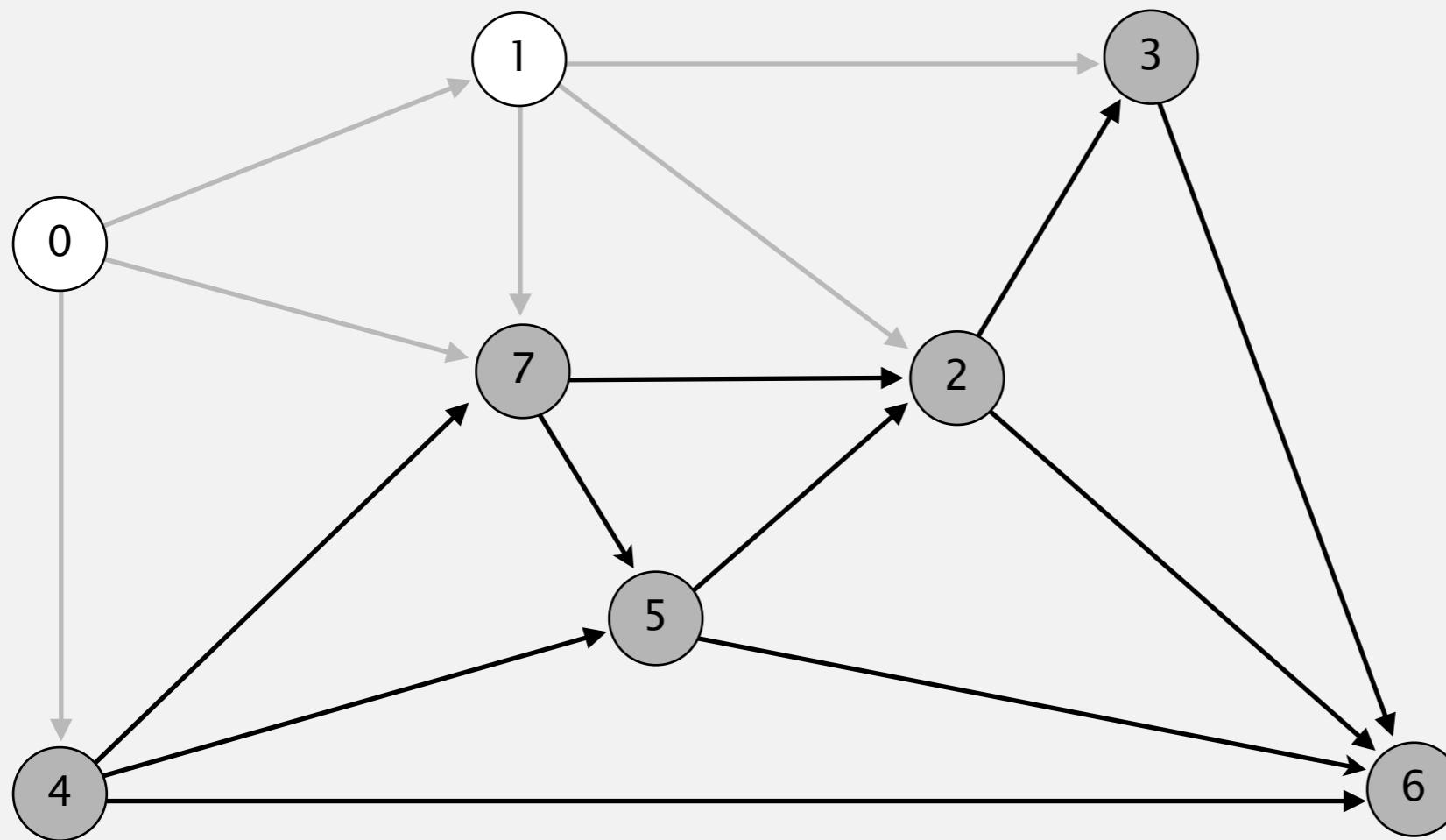
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 1

Dijkstra's algorithm demo

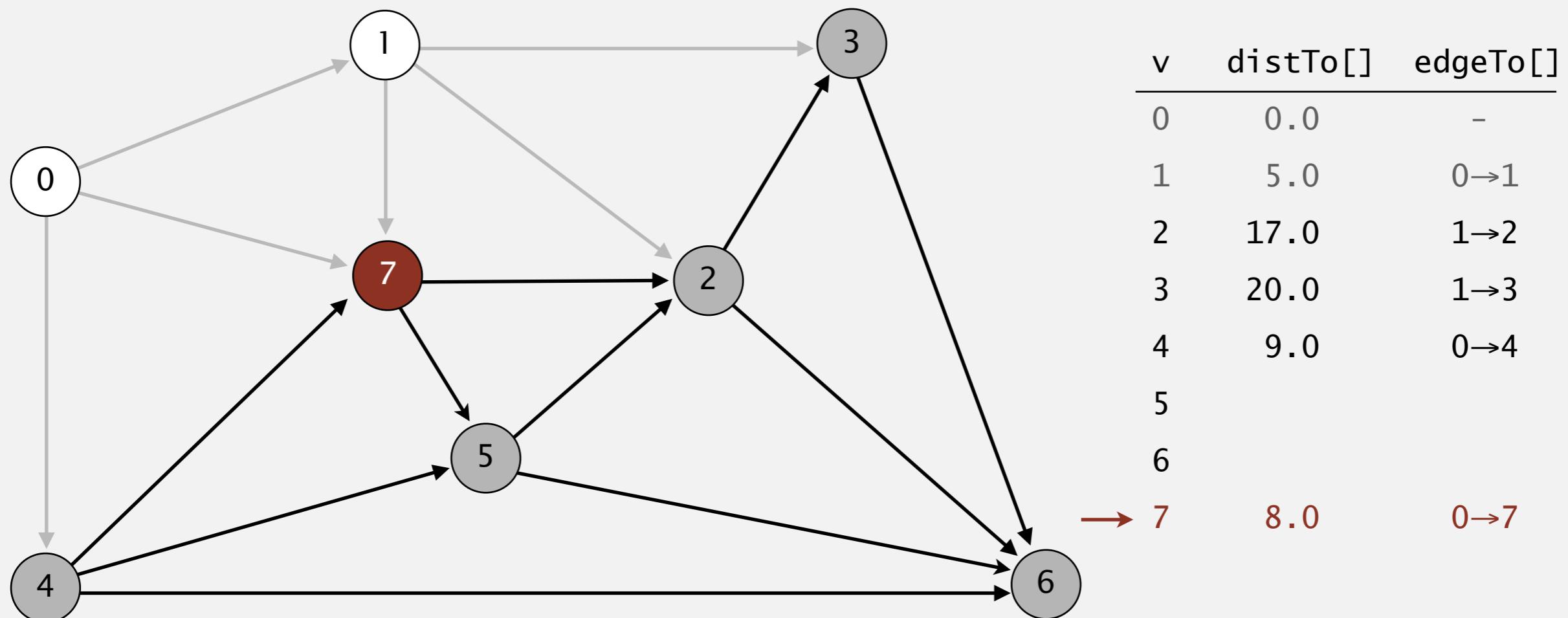
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

Dijkstra's algorithm demo

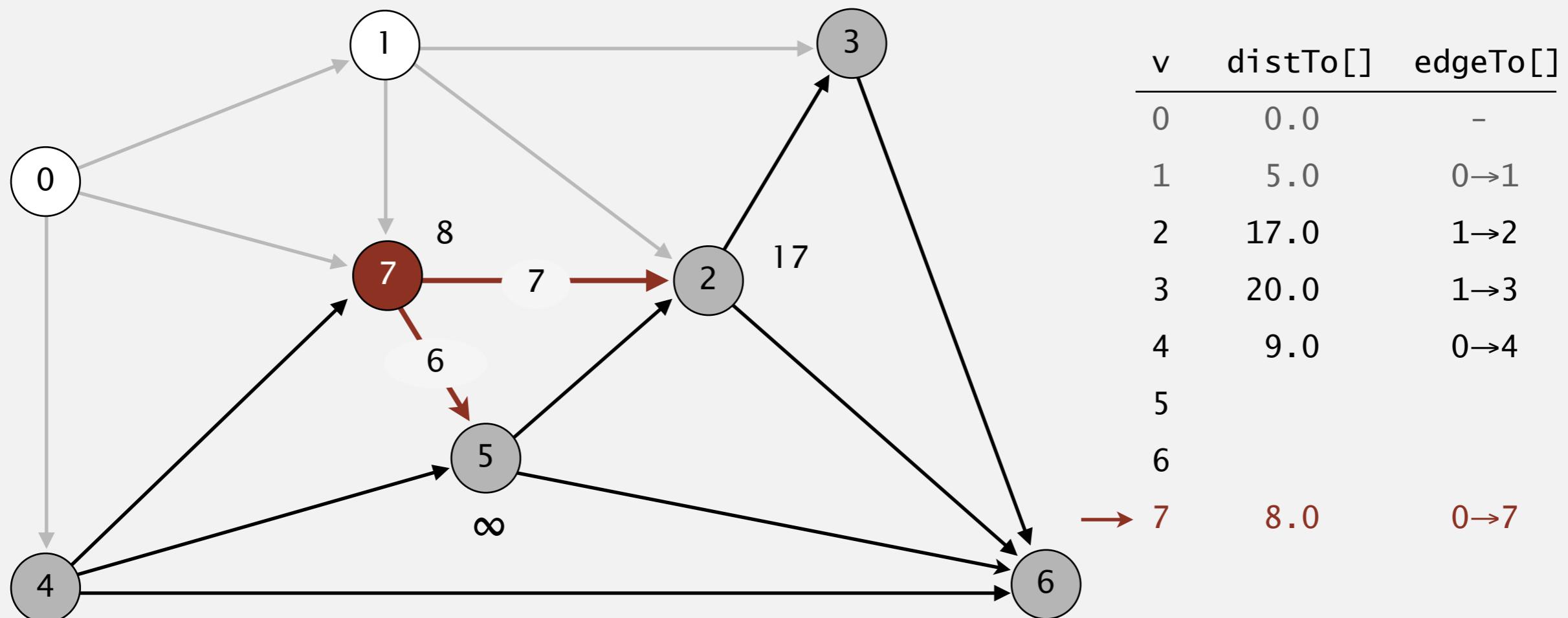
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



choose vertex 7

Dijkstra's algorithm demo

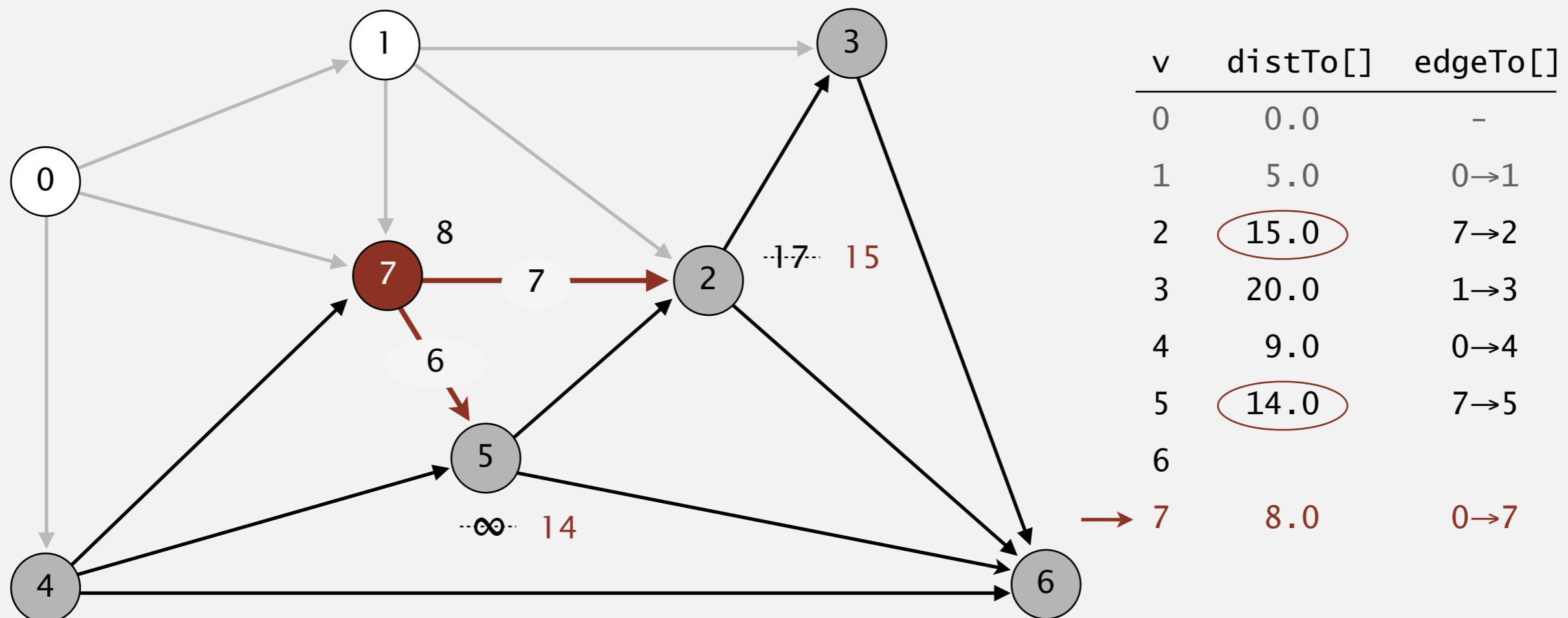
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 7

Dijkstra's algorithm demo

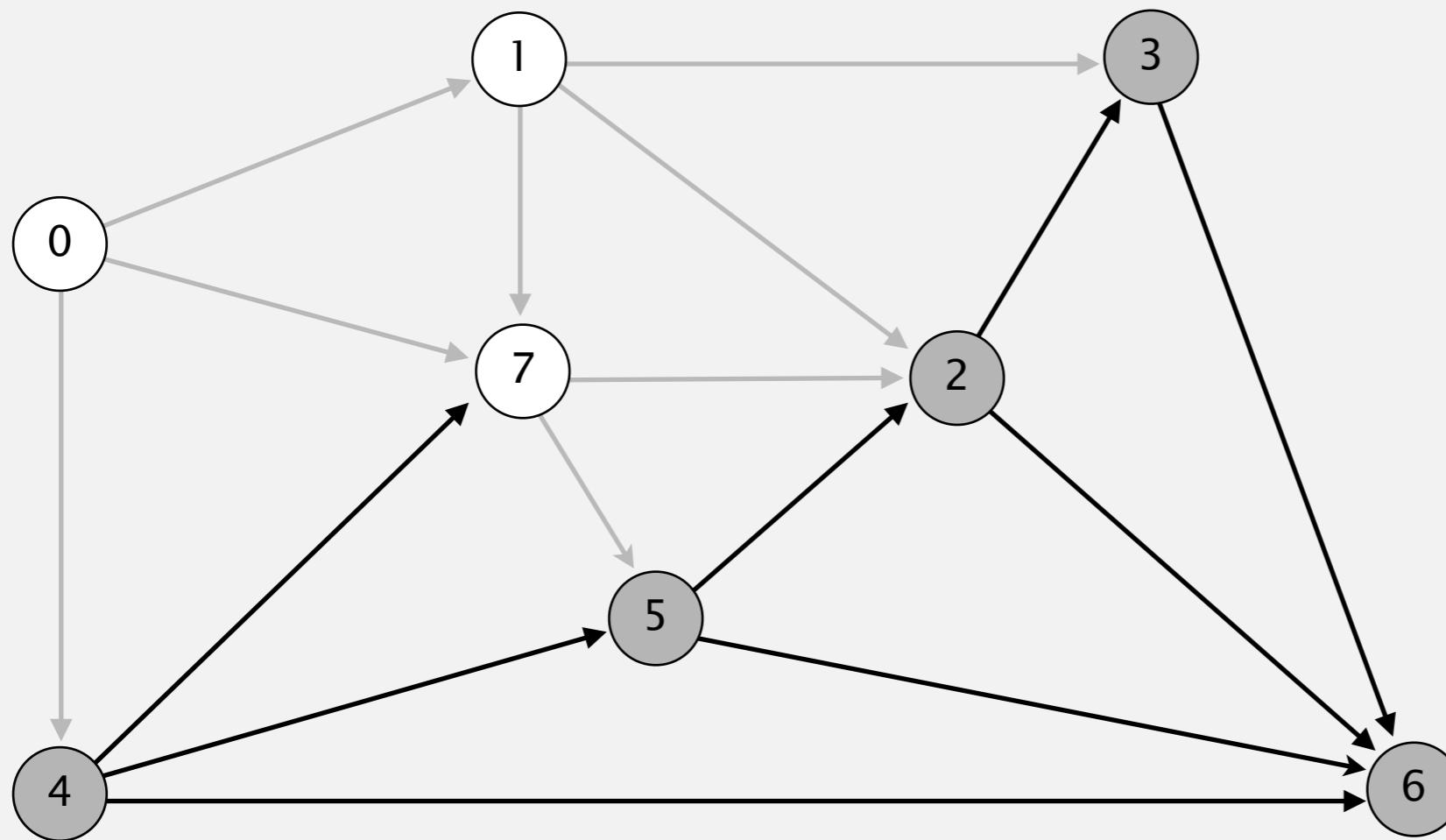
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 7

Dijkstra's algorithm demo

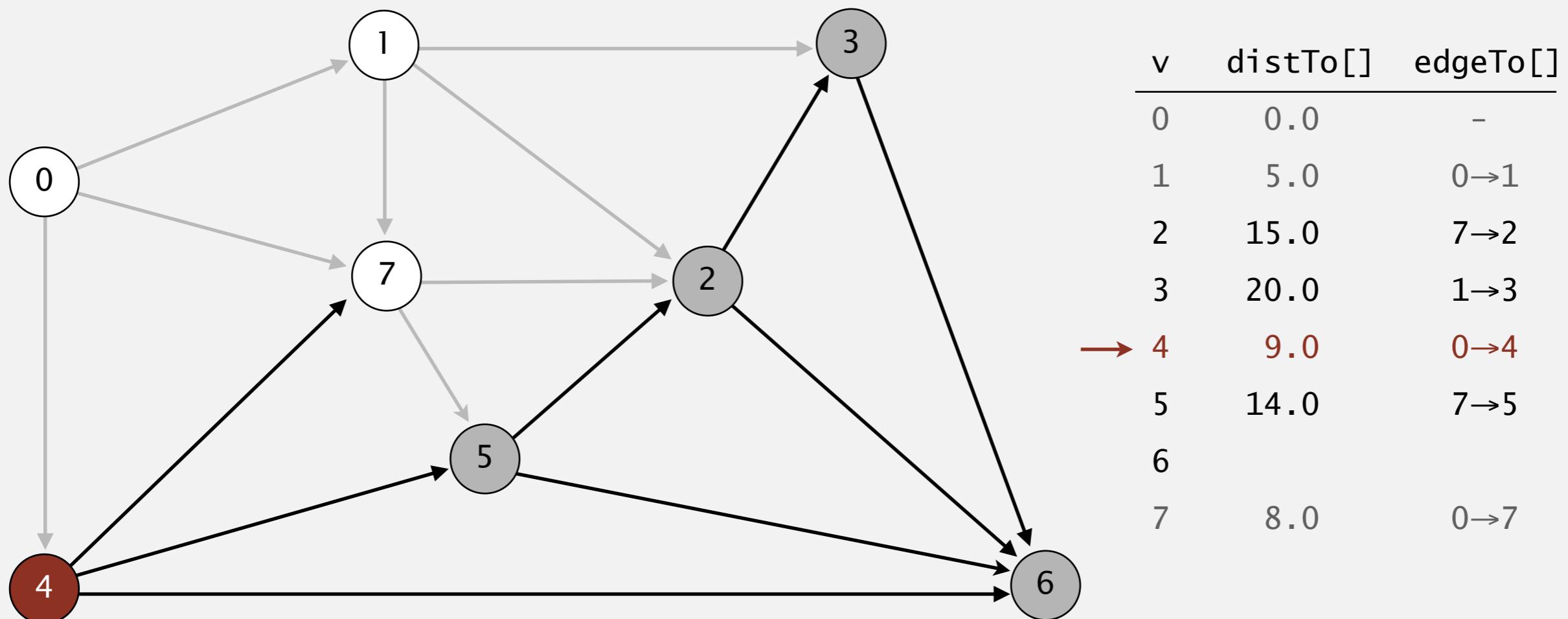
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	14.0	7→5
6		
7	8.0	0→7

Dijkstra's algorithm demo

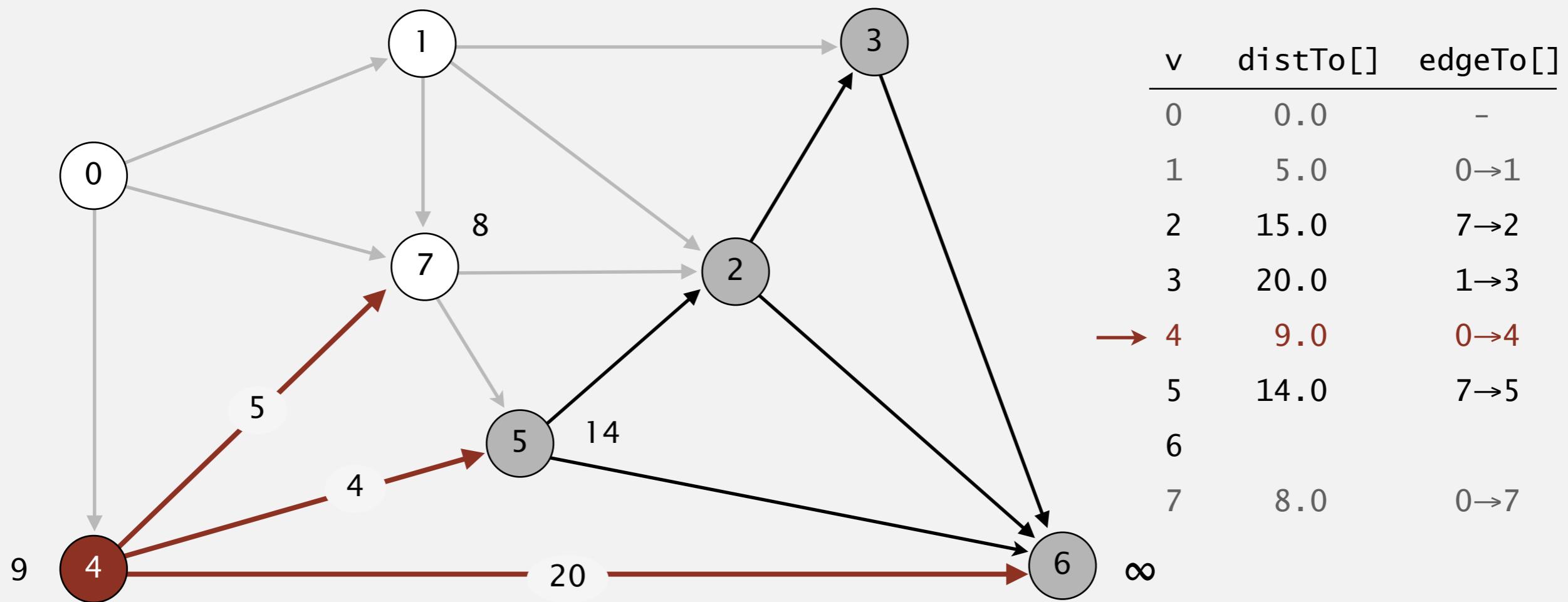
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



select vertex 4

Dijkstra's algorithm demo

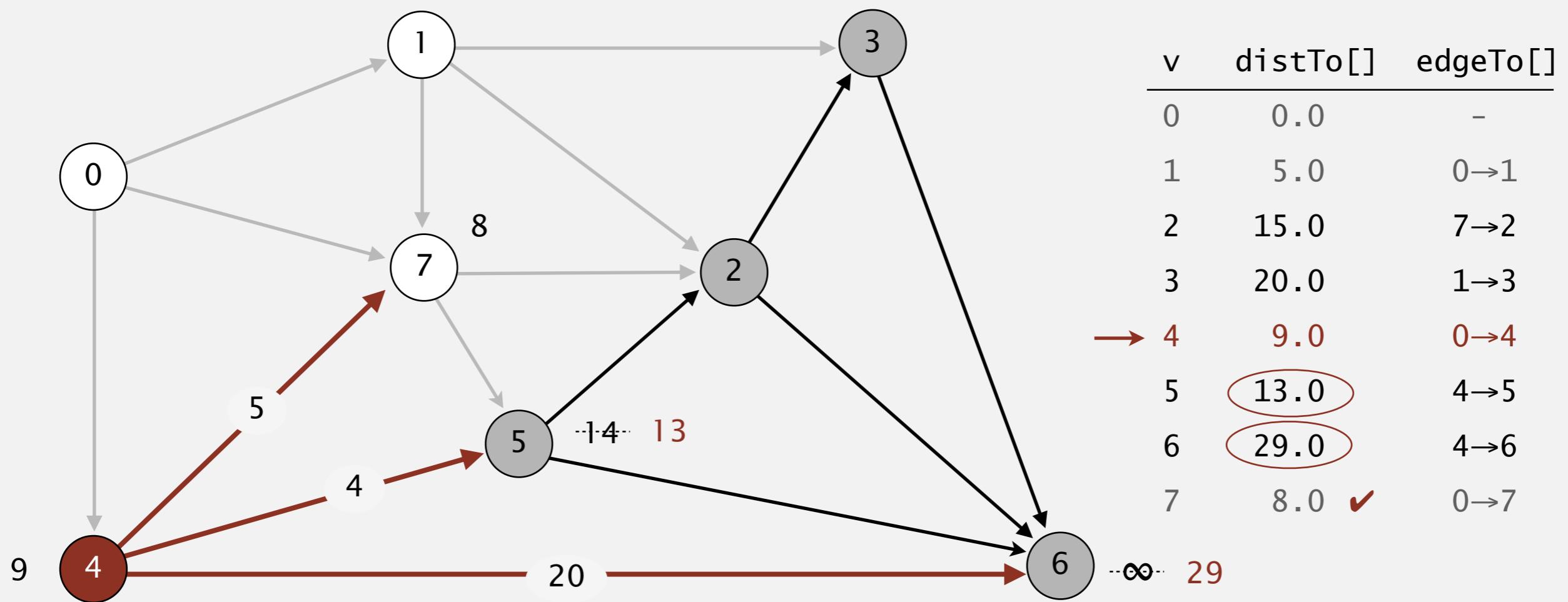
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 4

Dijkstra's algorithm demo

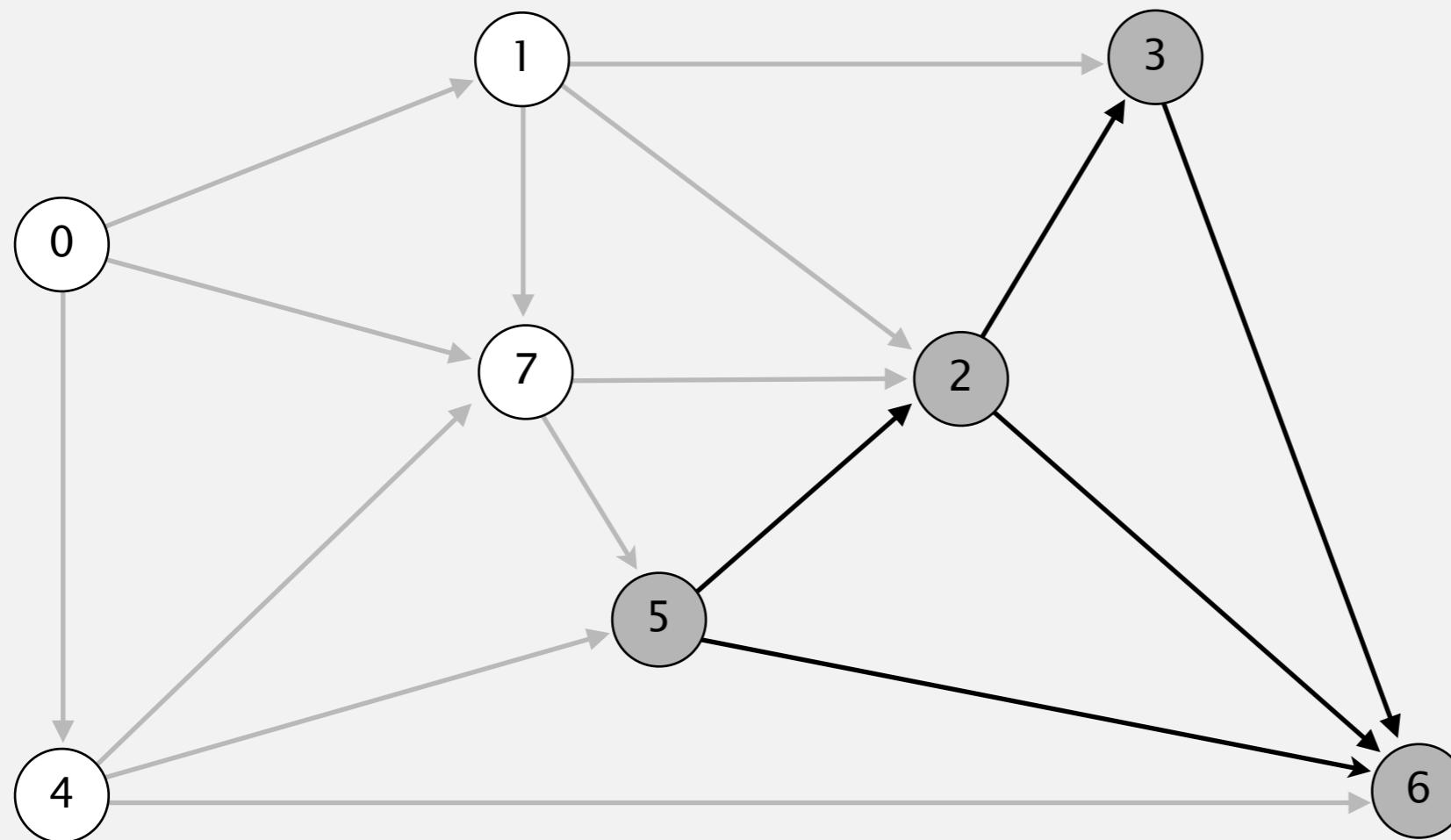
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 4

Dijkstra's algorithm demo

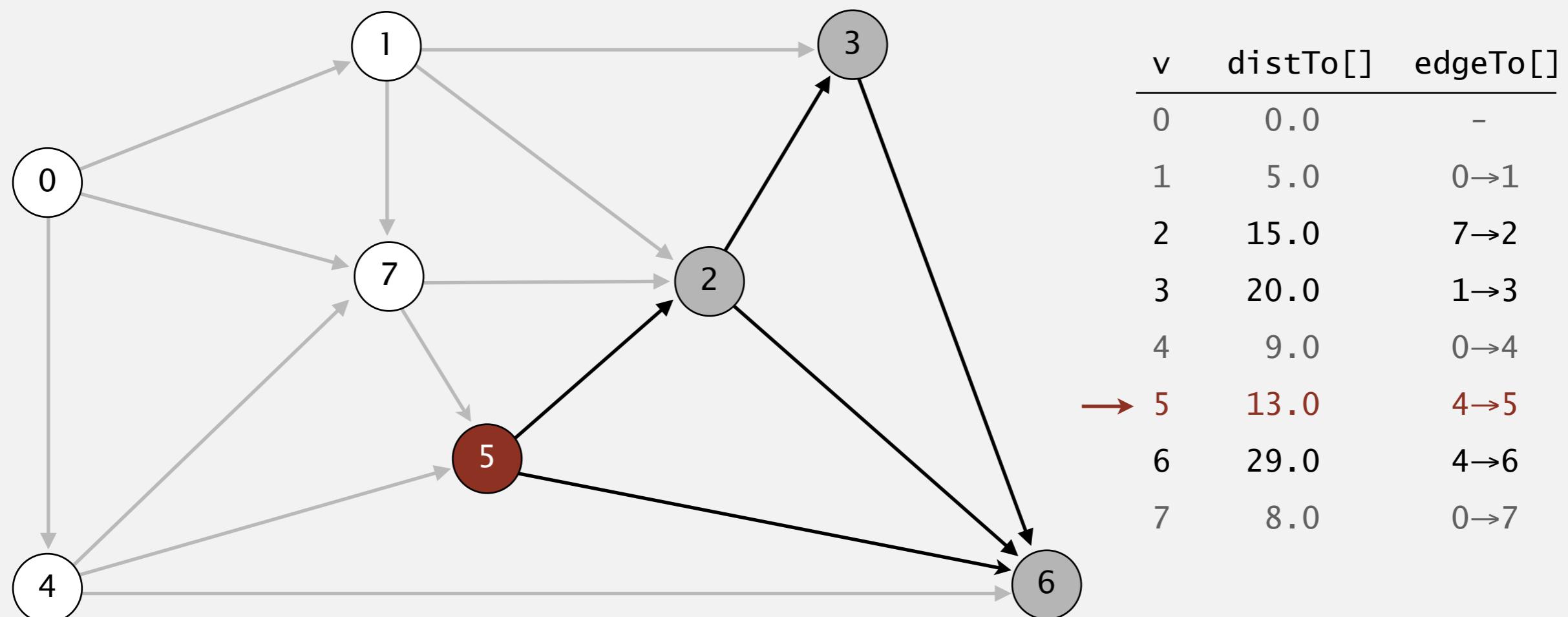
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Dijkstra's algorithm demo

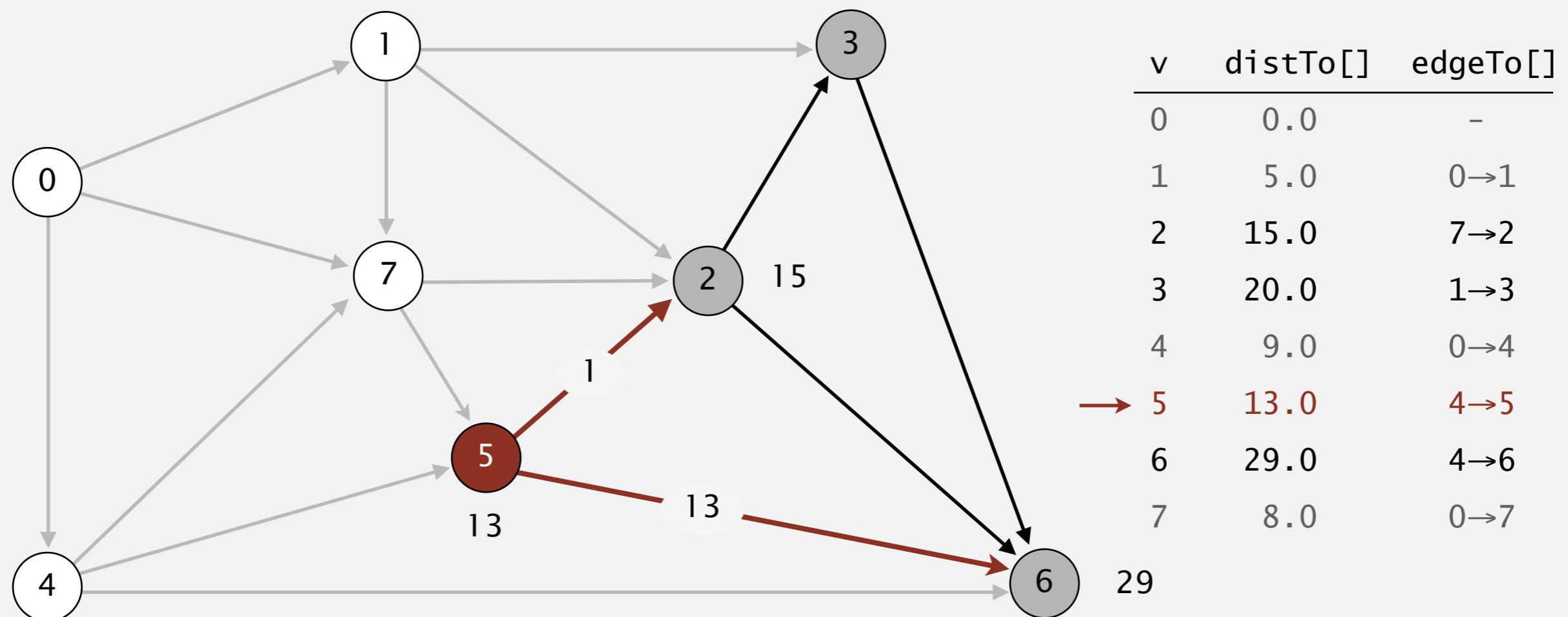
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



select vertex 5

Dijkstra's algorithm demo

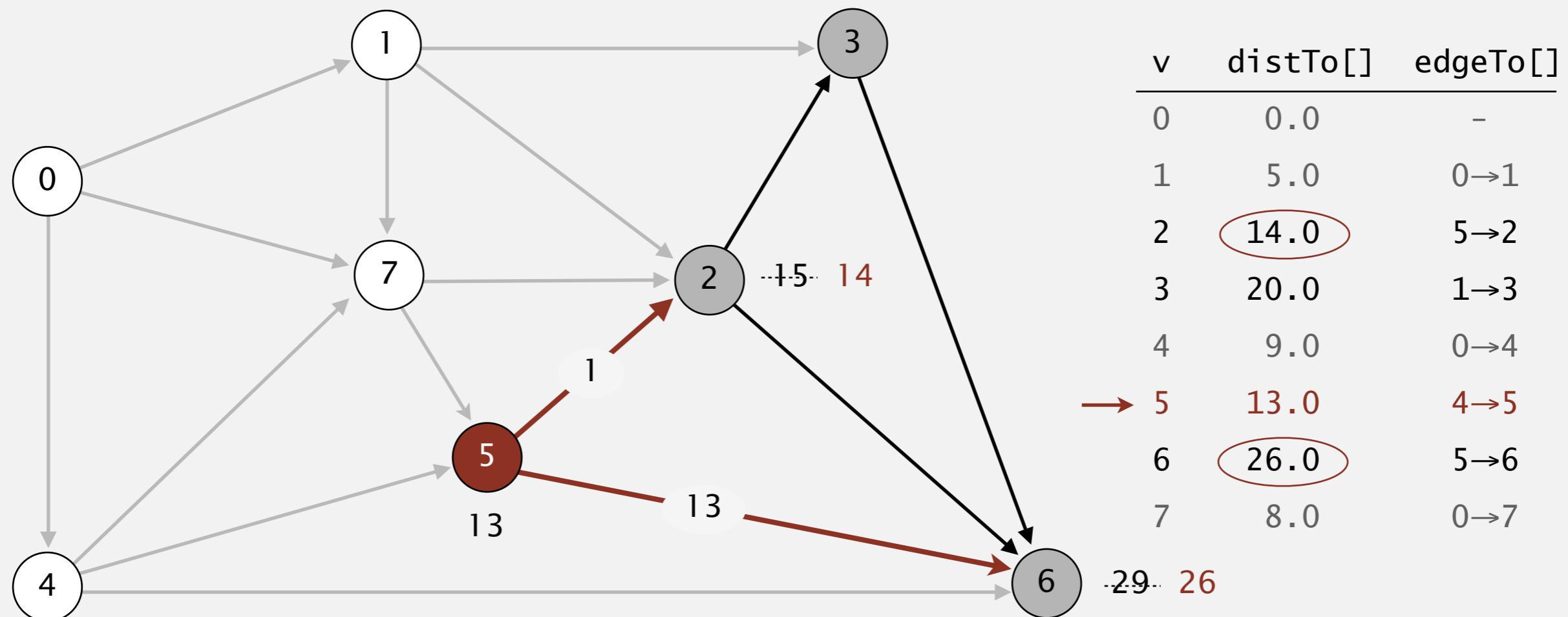
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 5

Dijkstra's algorithm demo

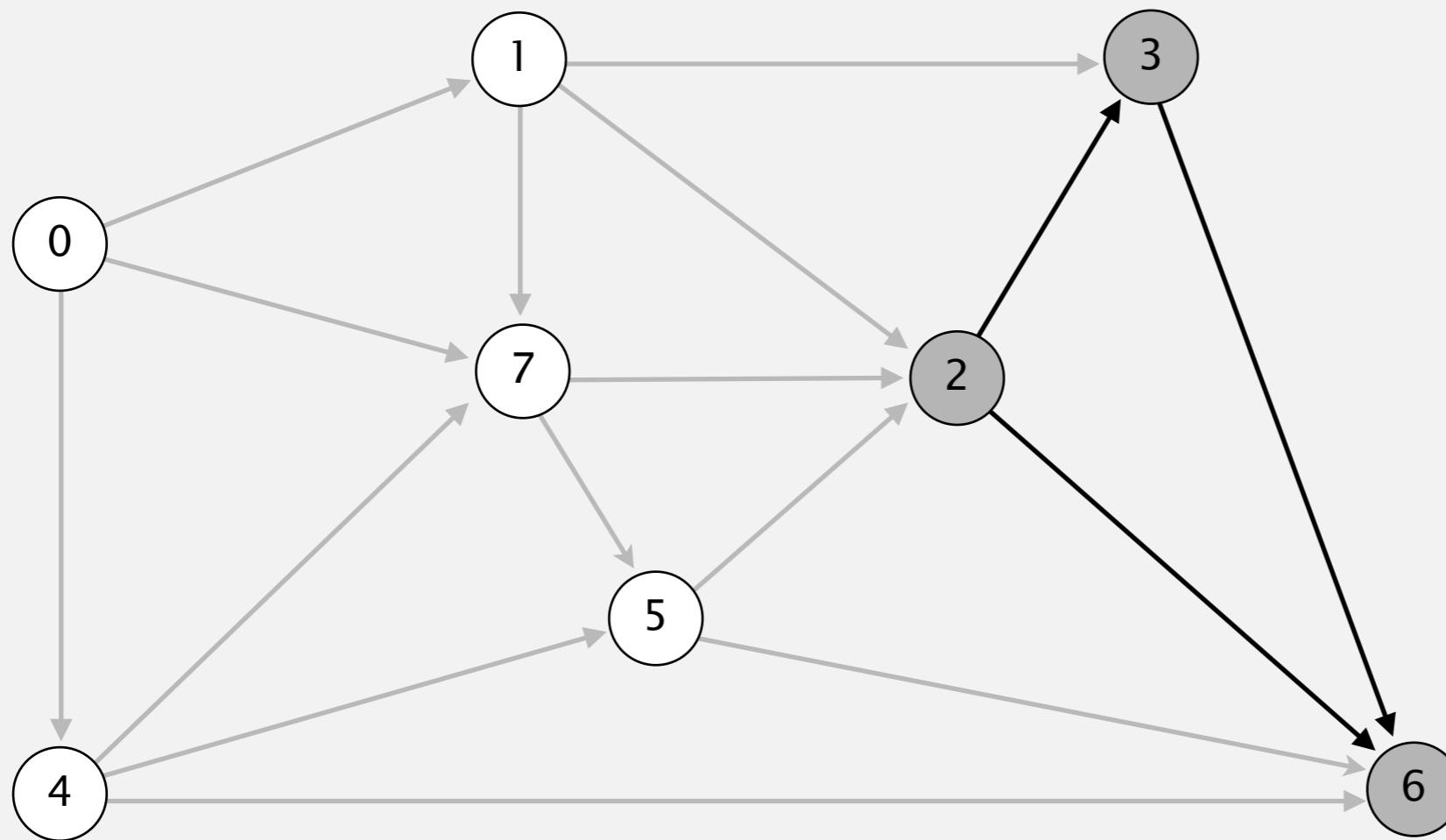
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 5

Dijkstra's algorithm demo

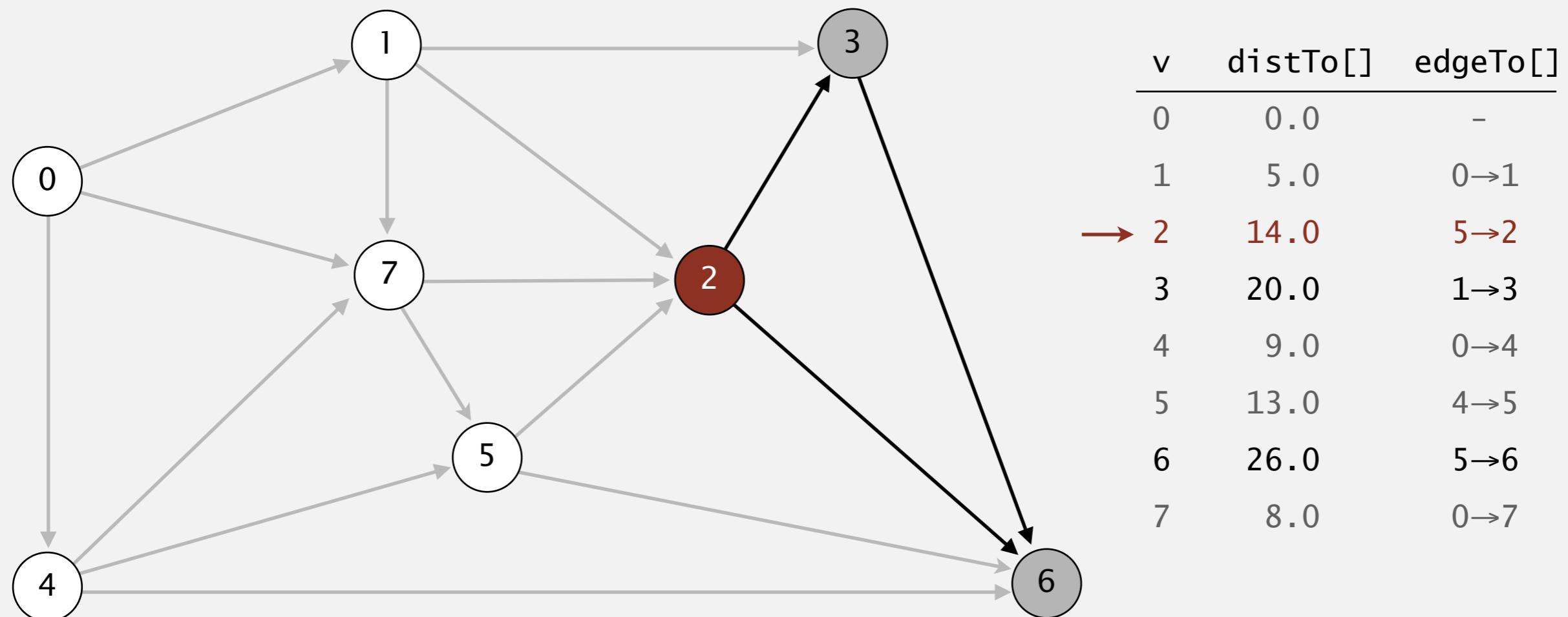
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Dijkstra's algorithm demo

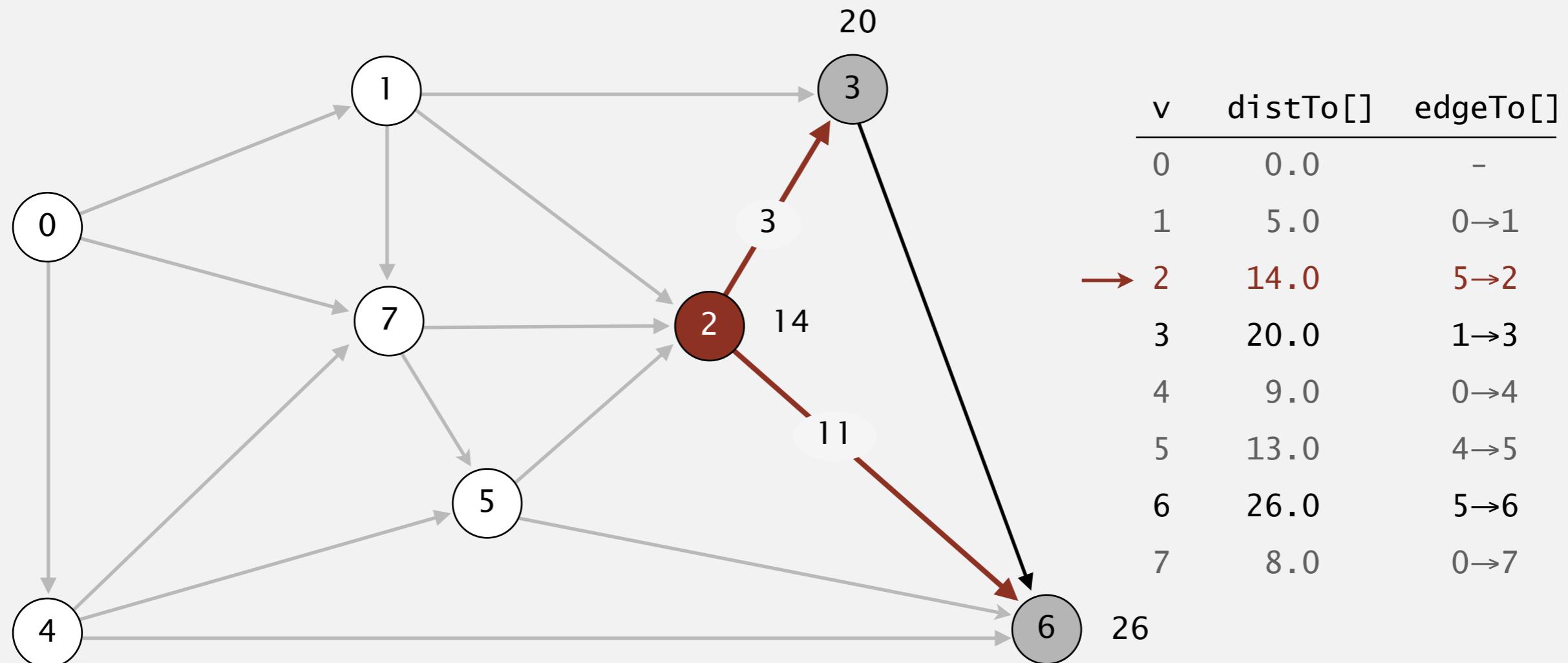
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



select vertex 2

Dijkstra's algorithm demo

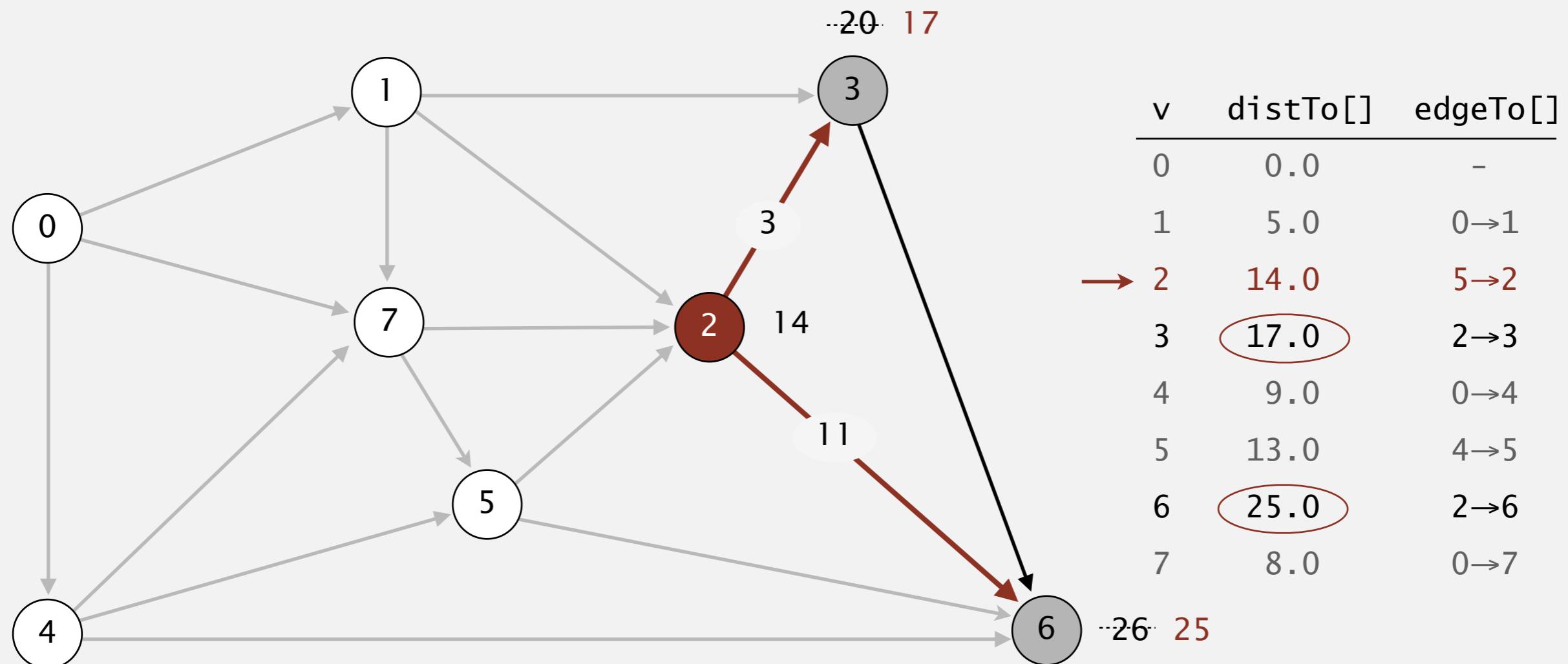
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 2

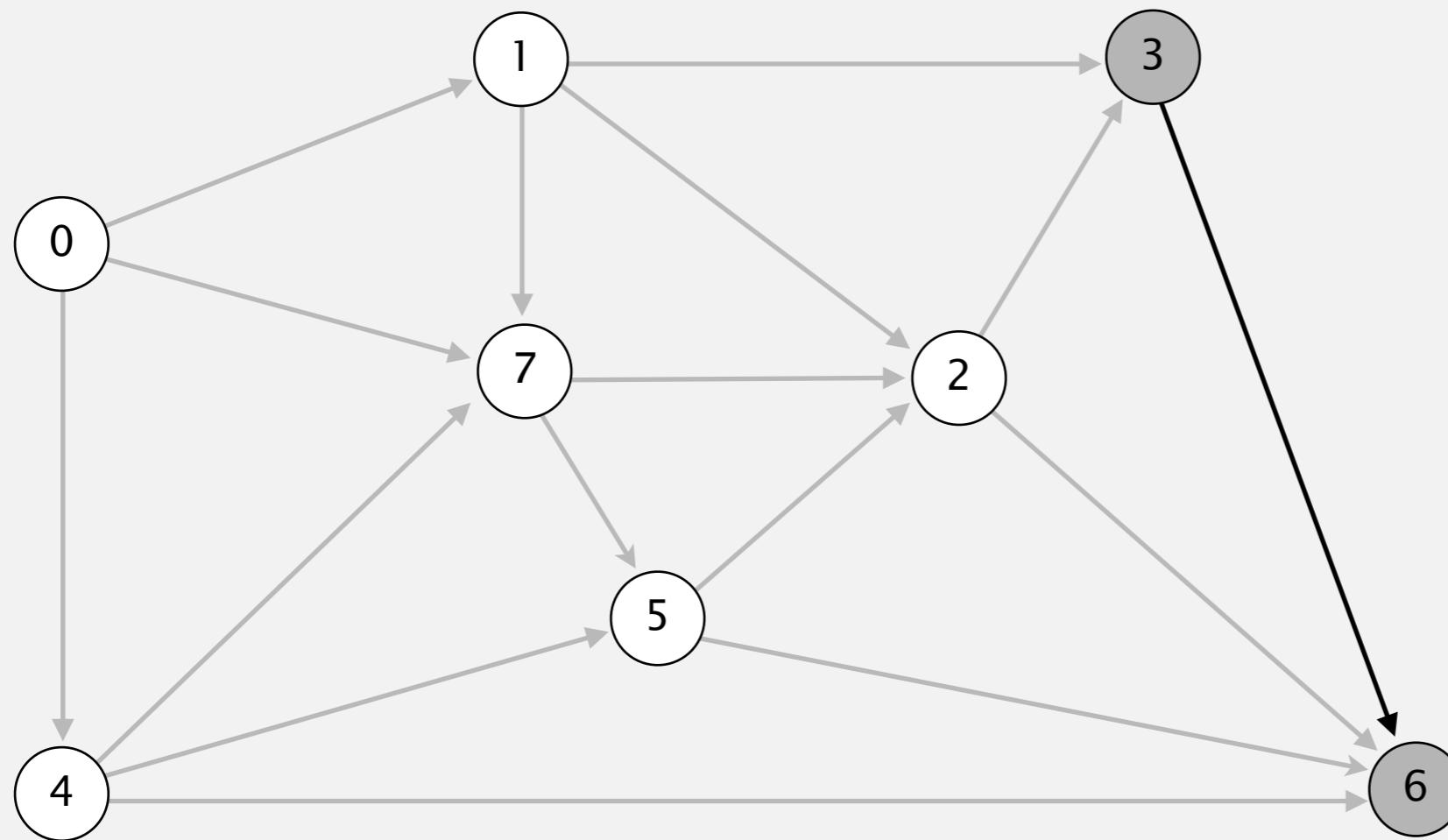
Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



Dijkstra's algorithm demo

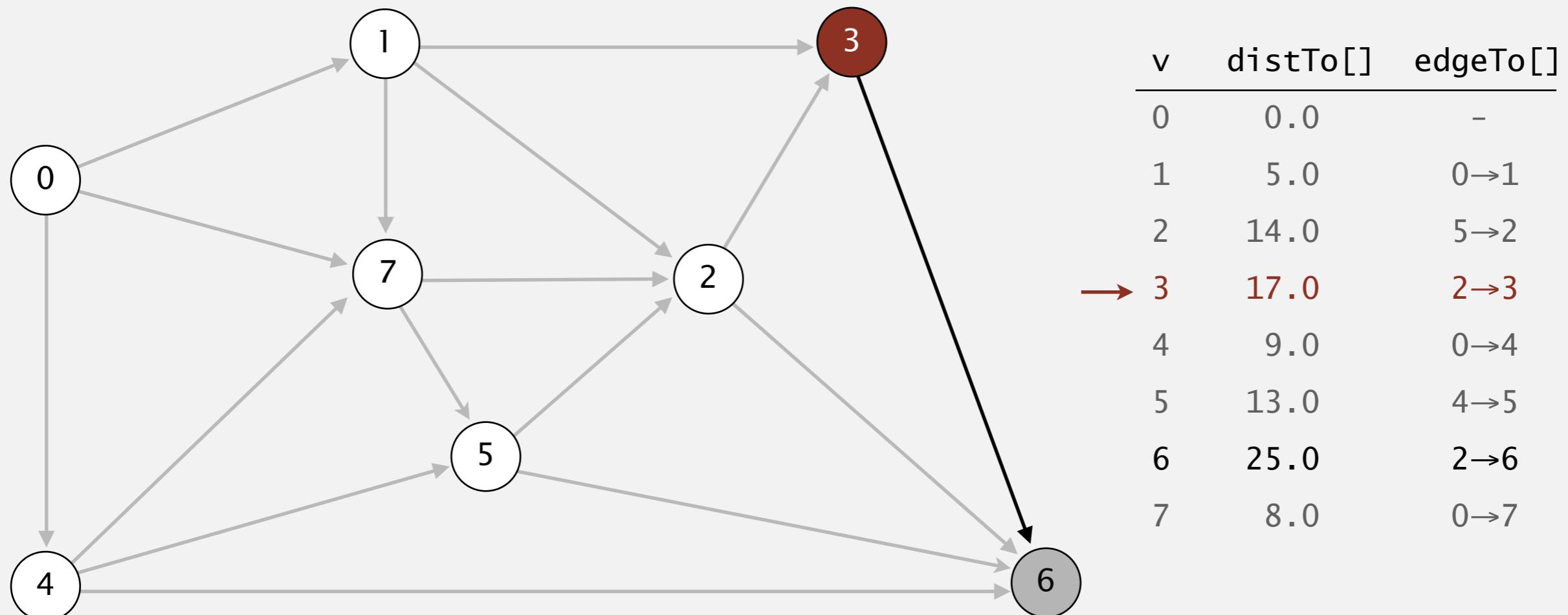
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Dijkstra's algorithm demo

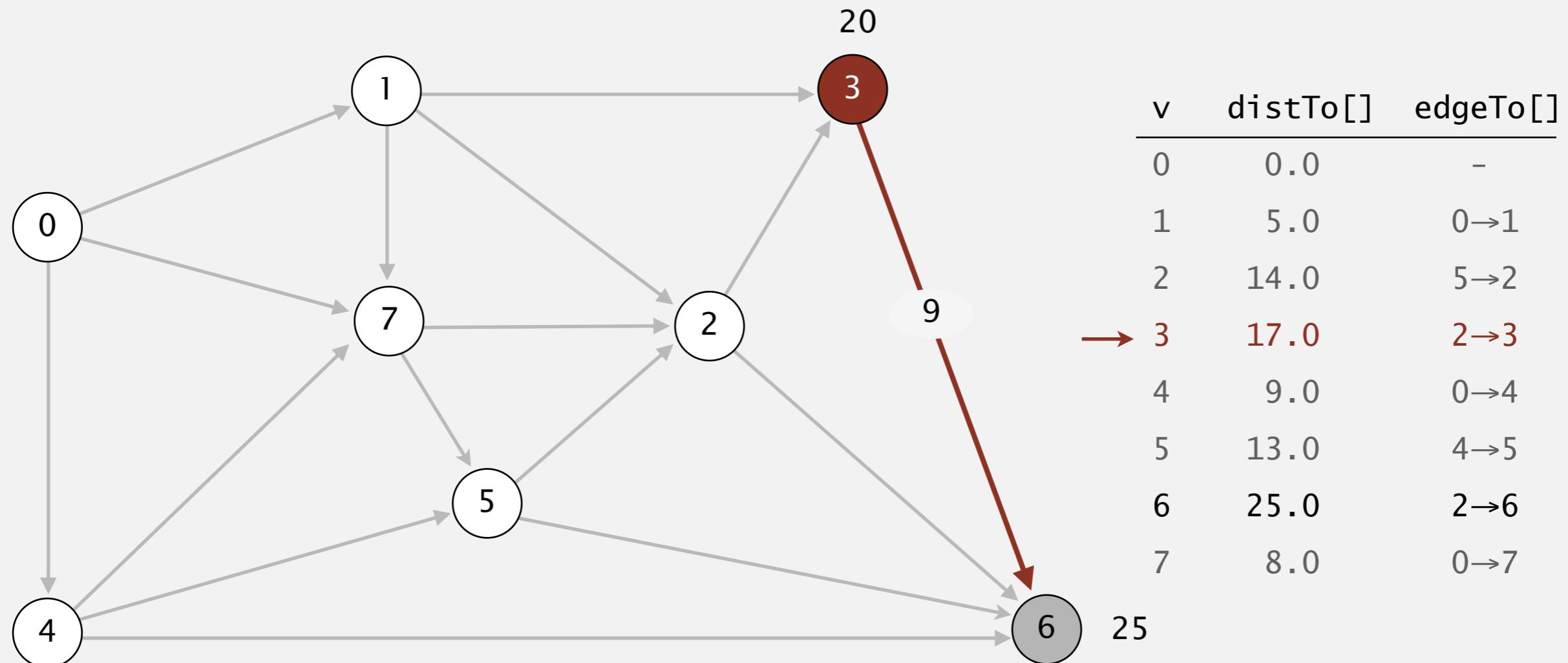
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



select vertex 3

Dijkstra's algorithm demo

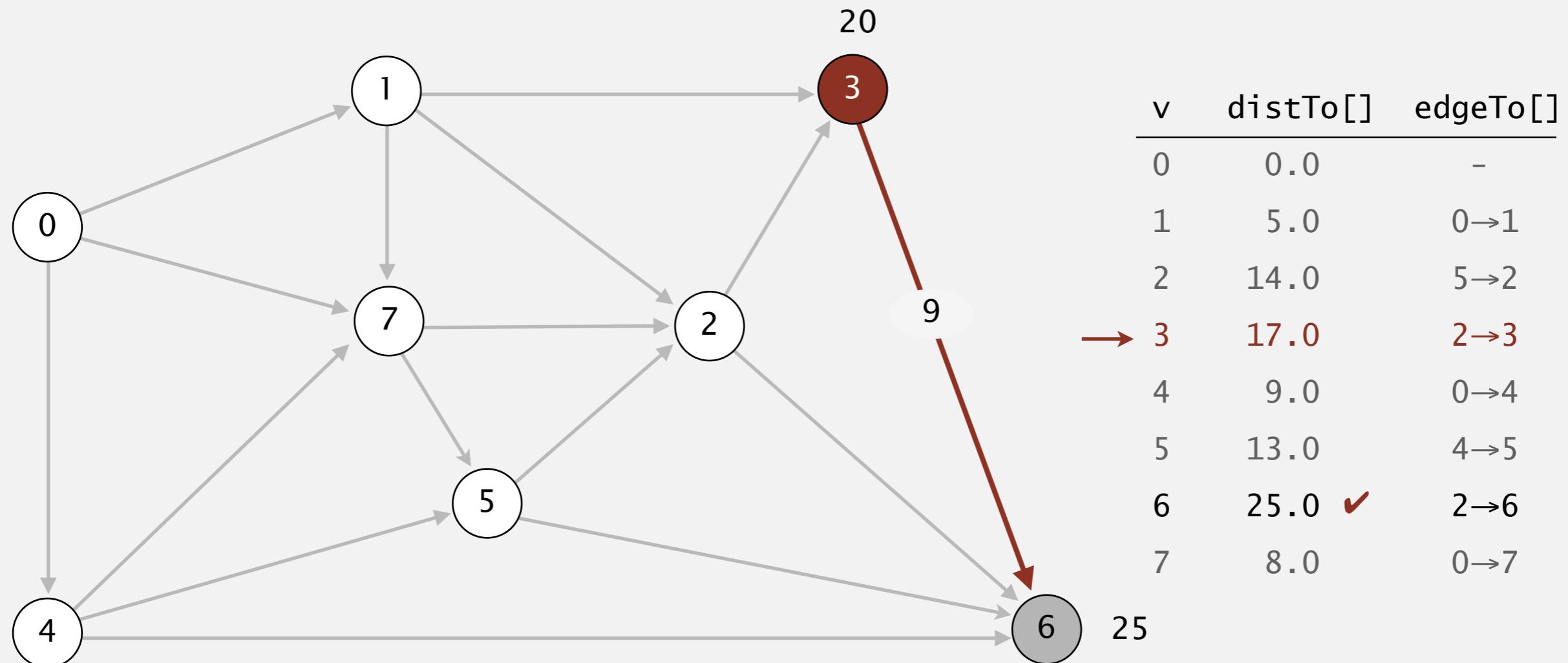
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 3

Dijkstra's algorithm demo

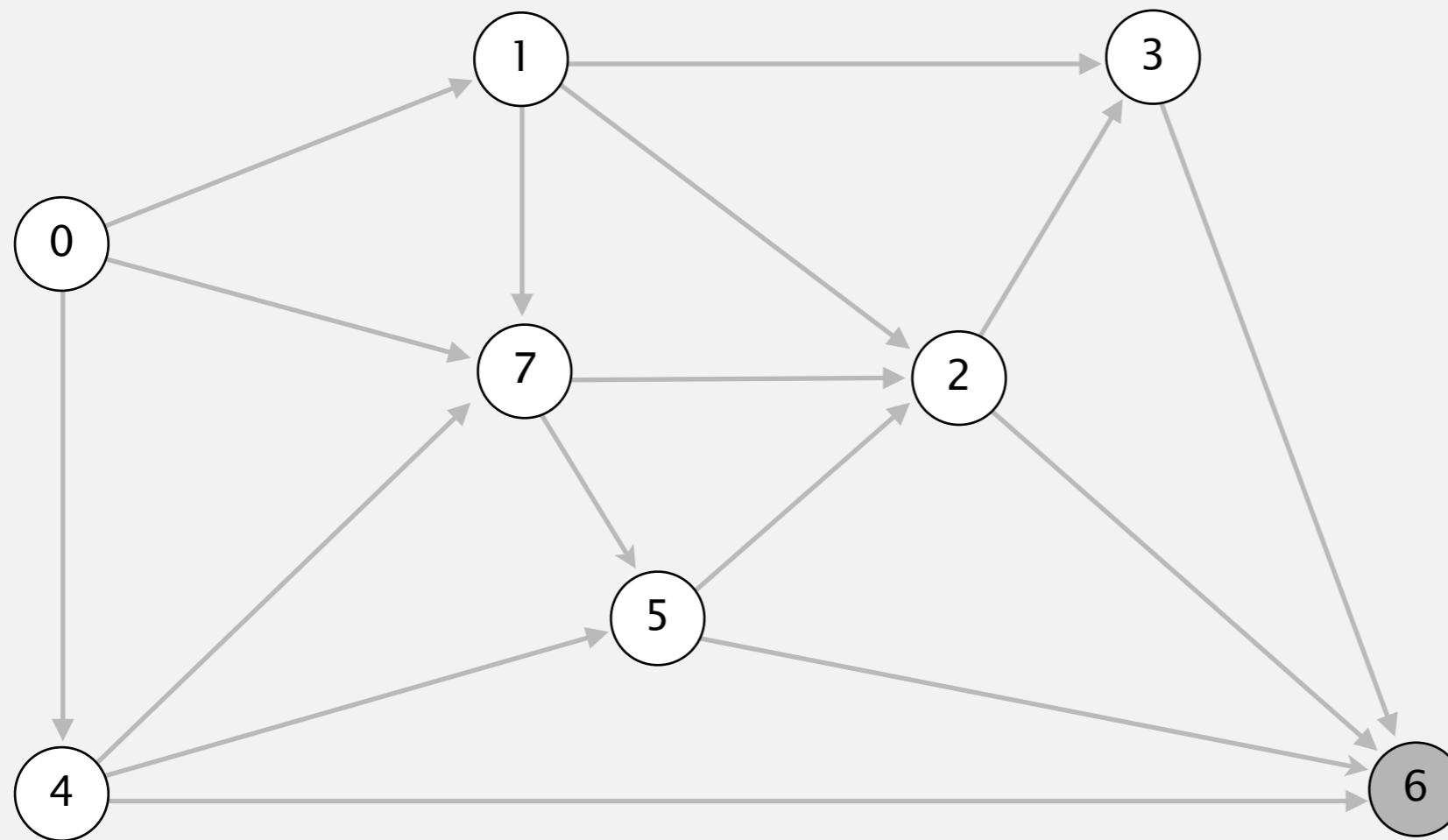
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 3

Dijkstra's algorithm demo

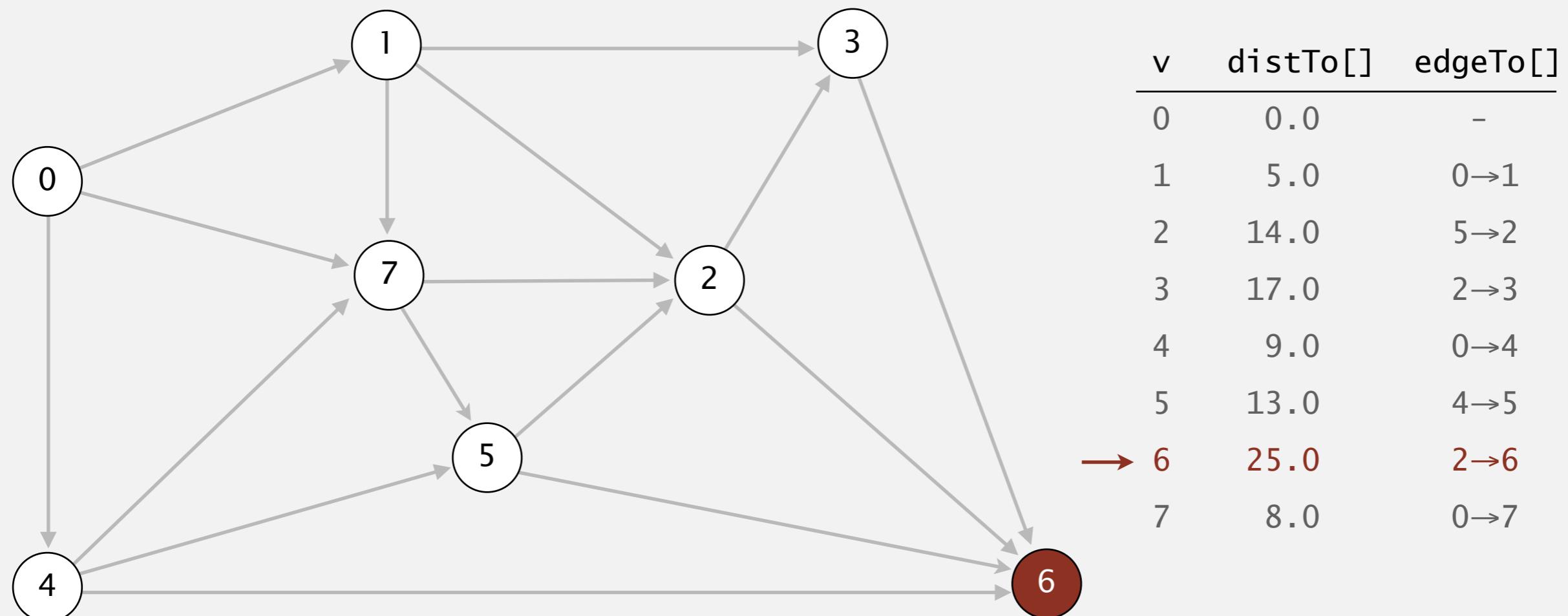
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Dijkstra's algorithm demo

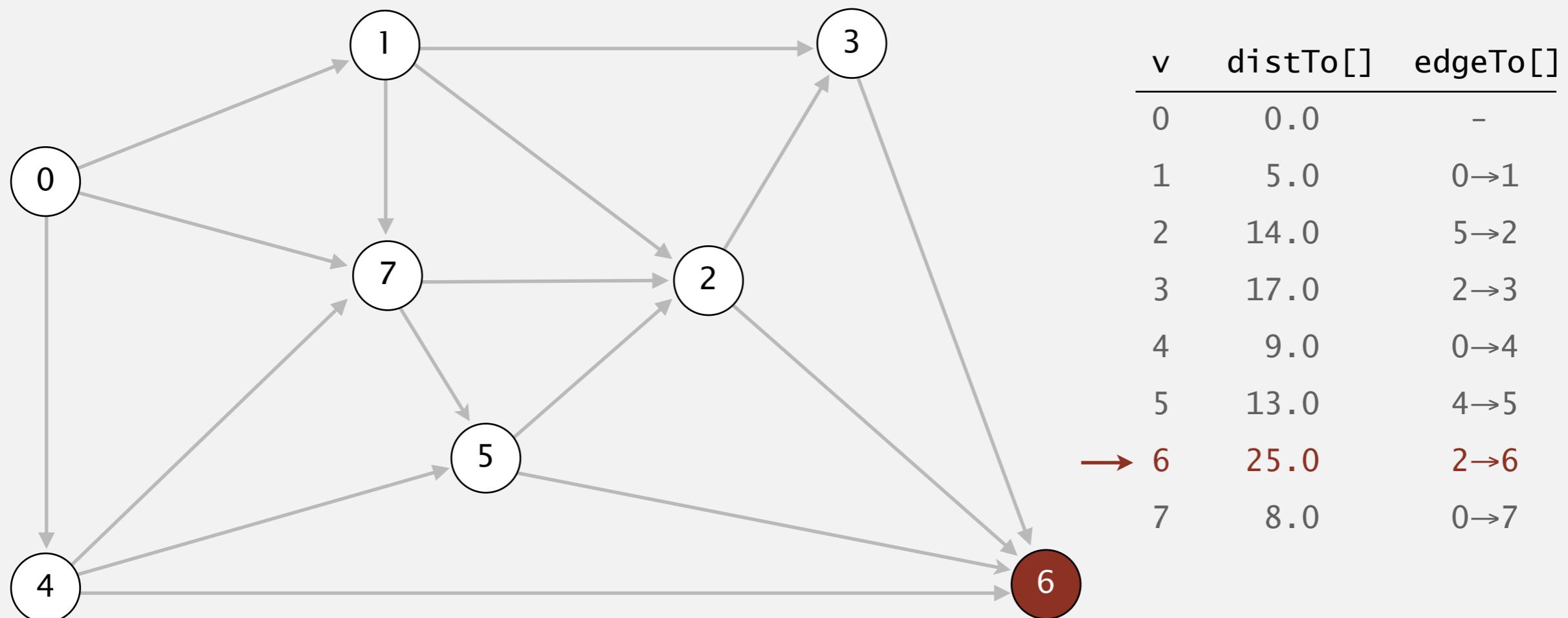
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



select vertex 6

Dijkstra's algorithm demo

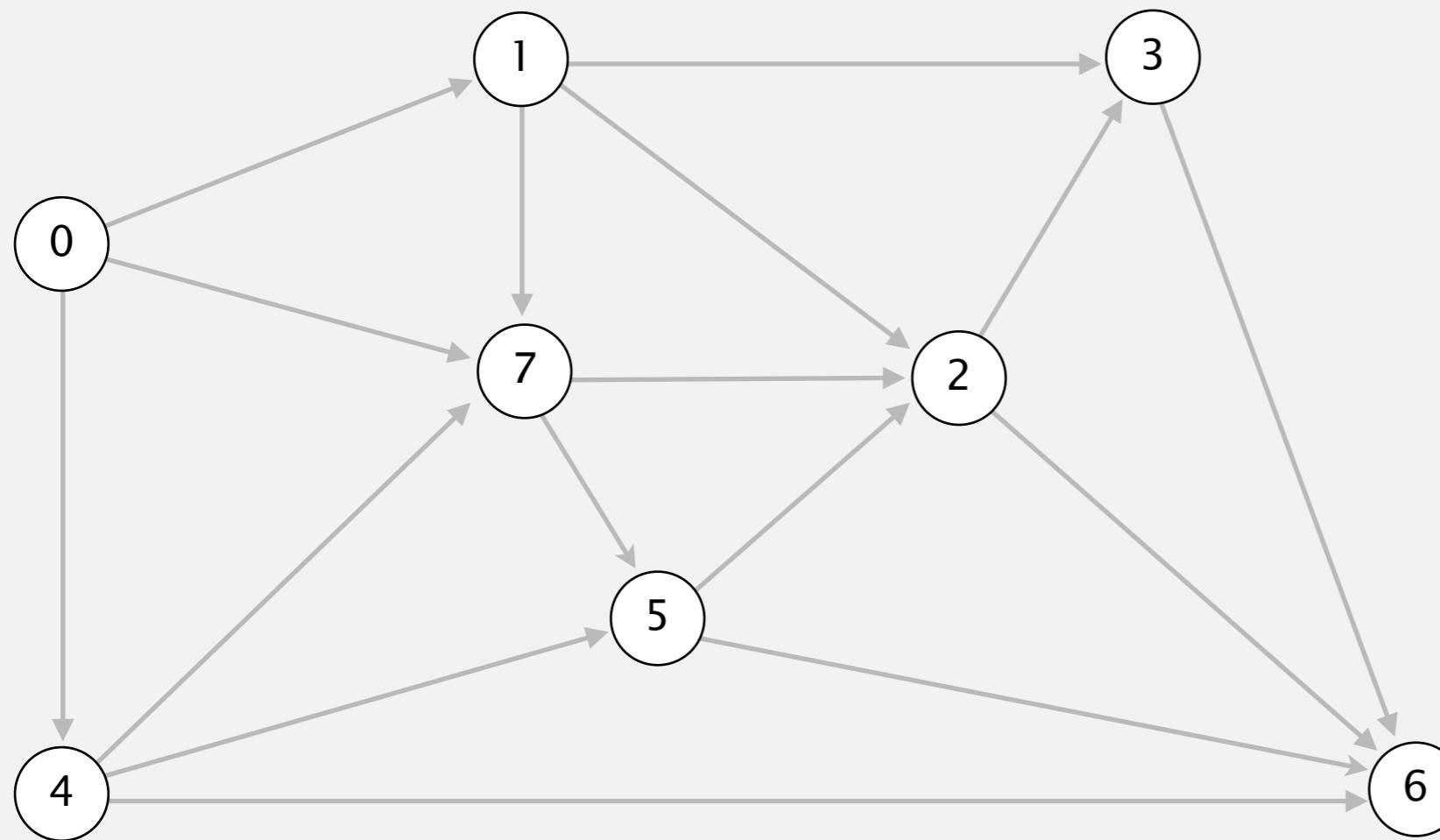
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



relax all edges pointing from 6

Dijkstra's algorithm demo

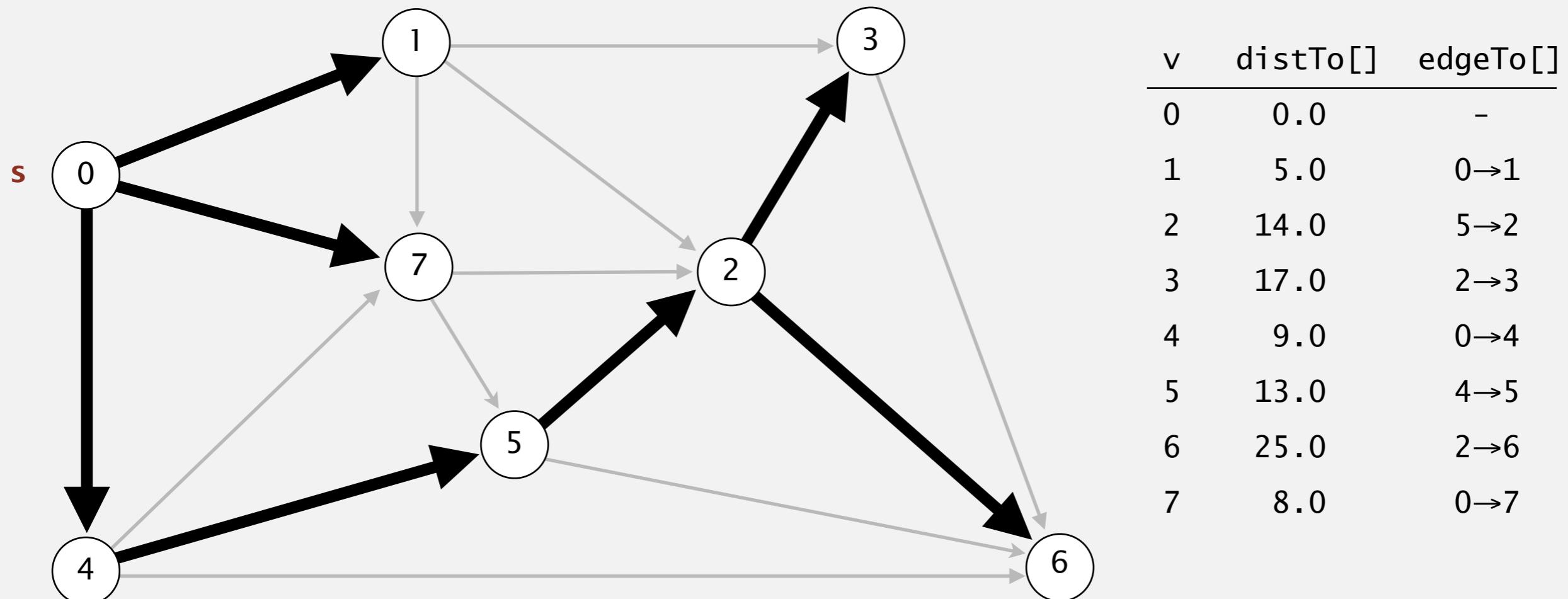
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



shortest-paths tree from vertex s

CSU22012: Data Structures and Algorithms II

Shortest paths in graphs

Ivana.Dusparic@scss.tcd.ie

Outline

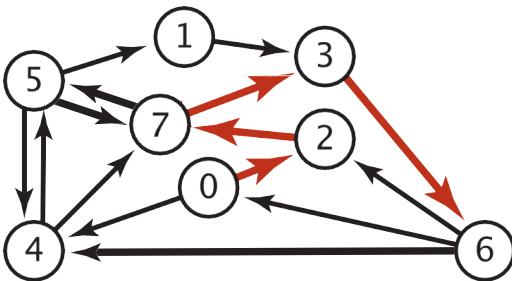
- › Shortest paths
 - Problem definition
 - Single source shortest path
 - › BFS?
 - › Topological sort – acyclic graphs but ok with negative weights
 - › Dijkstra – non negative weights but ok with cycles
 - › Bellman-Ford – non-negative cycles
 - Single-pair shortest path
 - › Uniform cost
 - › Greedy Best first
 - › A* search algorithm
 - All pairs shortest path
 - › Floyd-Warshall

Shortest paths in an edge-weighted digraph

Given an edge-weighted digraph, find the shortest path from s to t .

edge-weighted digraph

4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

Shortest path variants

Which vertices?

- Single source: from one vertex s to every other vertex.
- Single sink: from every vertex to one vertex t .
- Source-sink: from one vertex s to another t .
- All pairs: between all pairs of vertices.

Restrictions on edge weights?

- Nonnegative weights.

Cycles?

- No directed cycles.
- No negative cycles.

Properties

- › *Paths are directed.* A shortest path must respect the direction of its edges.
- › *The weights are not necessarily distances.* Geometric intuition can be helpful, but the edge weights might represent time or cost.
- › *Not all vertices need be reachable.* If t is not reachable from s , there is no path at all, and therefore there is no shortest path from s to t .
- › *Negative weights introduce complications.* For the moment, we assume that edge weights are positive (or zero).
- › *Shortest paths are normally simple.* Our algorithms ignore zero-weight edges that form cycles, so that the shortest paths they find have no cycles.
- › *Shortest paths are not necessarily unique.* There may be multiple paths of the lowest weight from one vertex to another; we are content to find any one of them.
- › *Parallel edges and self-loops may be present.* In the text, we assume that parallel edges are not present and use the notation $v \rightarrow w$ to refer to the edge from v to w

Types of edges recap

- › Undirected
 - DFS, BFS
- › Directed
 - DFS, BFS
- › Weighted undirected
 - MSTs
- › Here: Weighted directed

Weighted edge API

Edge weighted graph API

```
public class EdgeWeightedDigraph
```

EdgeWeightedDigraph(int V)	<i>empty V-vertex digraph</i>
EdgeWeightedDigraph(In in)	<i>construct from in</i>
int V()	<i>number of vertices</i>
int E()	<i>number of edges</i>
void addEdge(DirectedEdge e)	<i>add e to this digraph</i>
Iterable<DirectedEdge> adj(int v)	<i>edges pointing from v</i>
Iterable<DirectedEdge> edges()	<i>all edges in this digraph</i>
String toString()	<i>string representation</i>

Shortest path API

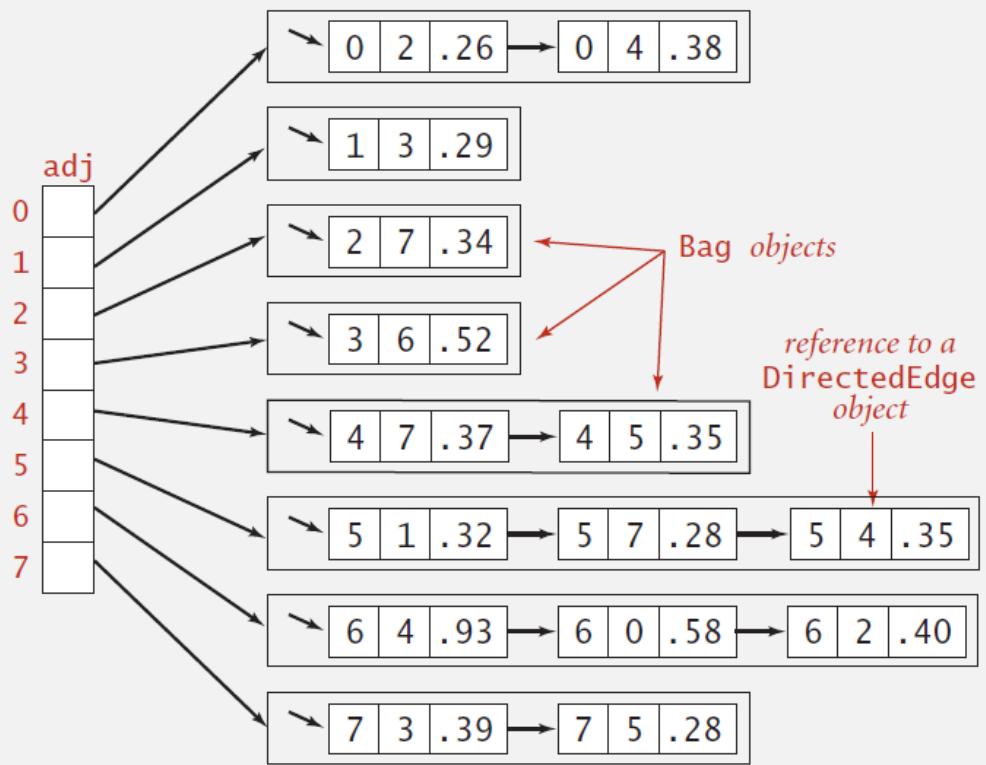
```
public class SP
{
    SP(EdgeWeightedDigraph G, int s) constructor
    double distTo(int v) distance from s to v, ∞ if no path
    boolean hasPathTo(int v) path from s to v?
    Iterable<DirectedEdge> pathTo(int v) path from s to v, null if none
}
```

Edge-weighted digraph: adjacency-lists representation

tinyEWD.txt

V → 8
E ← 15

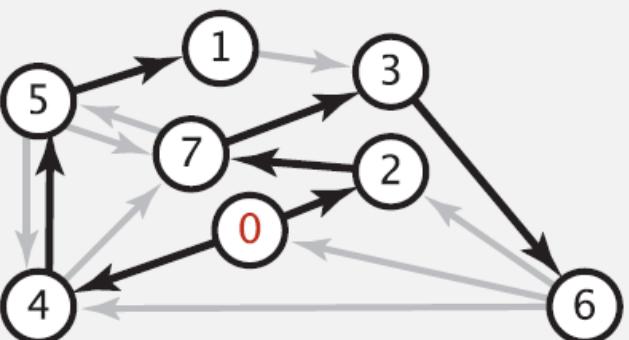
4	5	0.35
5	4	0.35
4	7	0.37
5	7	0.28
7	5	0.28
5	1	0.32
0	4	0.38
0	2	0.26
7	3	0.39
1	3	0.29
2	7	0.34
6	2	0.40
3	6	0.52
6	0	0.58
6	4	0.93



Data structures needed

› Two vertex-indexed arrays

- $\text{distTo}[v]$ is length of shortest path from s to v .
- $\text{edgeTo}[v]$ is last edge on shortest path from s to v .



shortest-paths tree from 0

	edgeTo[]	distTo[]
0	null	0
1	5->1	0.32
2	0->2	0.26
3	7->3	0.37
4	0->4	0.38
5	4->5	0.35
6	3->6	0.52
7	2->7	0.34

parent-link representation

Edge relaxation

- › Our shortest-paths implementations are based on an operation known as relaxation.
- › Initialize $\text{distTo}[s]$ to 0 and $\text{distTo}[v]$ to infinity for all other vertices v .
- › To relax an edge $v \rightarrow w$ means to test whether the best known way from s to w is to go from s to v , then take the edge from v to w , and, if so, update our data structures.

Edge relaxation

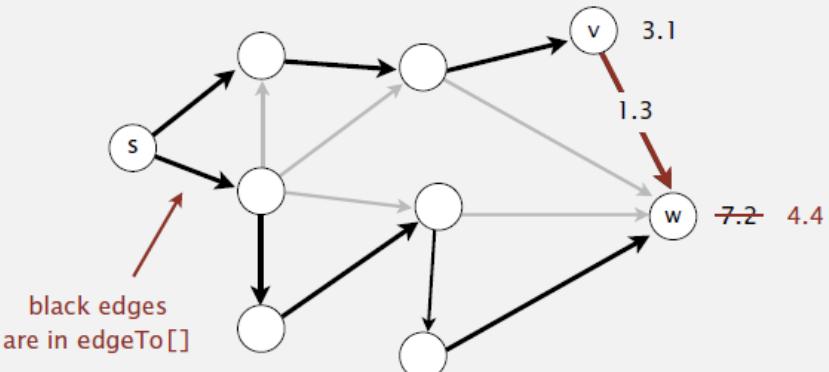
```
private void relax(DirectedEdge e) {  
    int v = e.from(), w = e.to();  
    if (distTo[w] > distTo[v] + e.weight()) {  
        distTo[w] = distTo[v] + e.weight();  
        edgeTo[w] = e;  
    }  
}
```

Edge relaxation

Relax edge $e = v \rightarrow w$.

- $\text{distTo}[v]$ is length of shortest known path from s to v .
- $\text{distTo}[w]$ is length of shortest known path from s to w .
- $\text{edgeTo}[w]$ is last edge on shortest known path from s to w .
- If $e = v \rightarrow w$ gives shorter path to w through v ,
update both $\text{distTo}[w]$ and $\text{edgeTo}[w]$.

$v \rightarrow w$ successfully relaxes



Vertex relaxation

- › *Vertex relaxation* - relax all the edges pointing from a given vertex.

```
private void relax(EdgeWeightedDigraph G, int v) {
    for (DirectedEdge e : G.adj(v)) {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
        }
    }
}
```

Shortest-paths optimality conditions

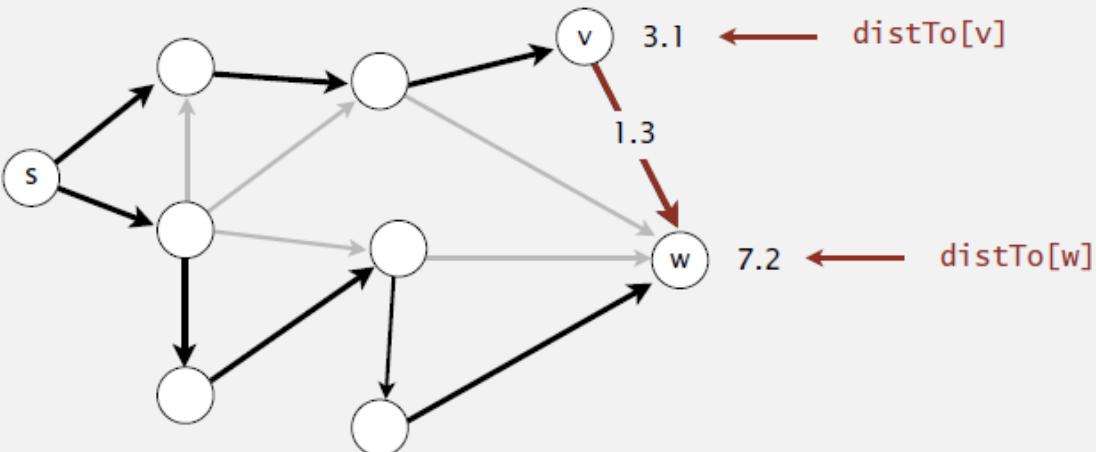
Proposition. Let G be an edge-weighted digraph.

Then $\text{distTo}[]$ are the shortest path distances from s iff:

- $\text{distTo}[s] = 0$.
- For each vertex v , $\text{distTo}[v]$ is the length of some path from s to v .
- For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.

Pf. \Leftarrow [necessary]

- Suppose that $\text{distTo}[w] > \text{distTo}[v] + e.\text{weight}()$ for some edge $e = v \rightarrow w$.
- Then, e gives a path from s to w (through v) of length less than $\text{distTo}[w]$.



Shortest-paths optimality conditions

Proposition. Let G be an edge-weighted digraph.

Then $\text{distTo}[]$ are the shortest path distances from s iff:

- $\text{distTo}[s] = 0$.
- For each vertex v , $\text{distTo}[v]$ is the length of some path from s to v .
- For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.

Pf. \Rightarrow [sufficient]

- Suppose that $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k = w$ is a shortest path from s to w .
- Then, $\text{distTo}[v_1] \leq \text{distTo}[v_0] + e_1.\text{weight}()$
 $\text{distTo}[v_2] \leq \text{distTo}[v_1] + e_2.\text{weight}()$
 \dots
 $\text{distTo}[v_k] \leq \text{distTo}[v_{k-1}] + e_k.\text{weight}()$

$e_i = i^{\text{th}}$ edge on shortest path from s to w

- Add inequalities; simplify; and substitute $\text{distTo}[v_0] = \text{distTo}[s] = 0$:

$$\text{distTo}[w] = \text{distTo}[v_k] \leq \underline{e_1.\text{weight}() + e_2.\text{weight}() + \dots + e_k.\text{weight}()}$$

weight of shortest path from s to w

- Thus, $\text{distTo}[w]$ is the weight of shortest path to w . ■

$\overleftarrow{\text{weight of some path from } s \text{ to } w}$

Generic shortest path algorithm

Generic algorithm (to compute SPT from s)

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat until optimality conditions are satisfied:

- Relax any edge.

Proposition. Generic algorithm computes SPT (if it exists) from s.

Pf sketch.

- The entry $\text{distTo}[v]$ is always the length of a simple path from s to v.
- Each successful relaxation decreases $\text{distTo}[v]$ for some v.
- The entry $\text{distTo}[v]$ can decrease at most a finite number of times. ■

Generic shortest path algorithm

Efficient implementations. How to choose which edge to relax?

- Ex 1. Dijkstra's algorithm (nonnegative weights).
- Ex 2. Topological sort algorithm (no directed cycles).
- Ex 3. Bellman-Ford algorithm (no negative cycles).

Dijkstra's shortest path algorithm

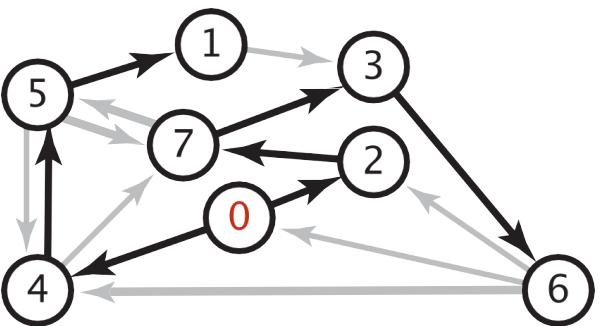
Data structures for single-source shortest paths

Goal. Find the shortest path from s to every other vertex.

Observation. A **shortest-paths tree** (SPT) solution exists.

Consequence. Can represent the SPT with two vertex-indexed arrays:

- $\text{distTo}[v]$ is length of shortest path from s to v .
- $\text{edgeTo}[v]$ is last edge on shortest path from s to v .



shortest-paths tree from 0

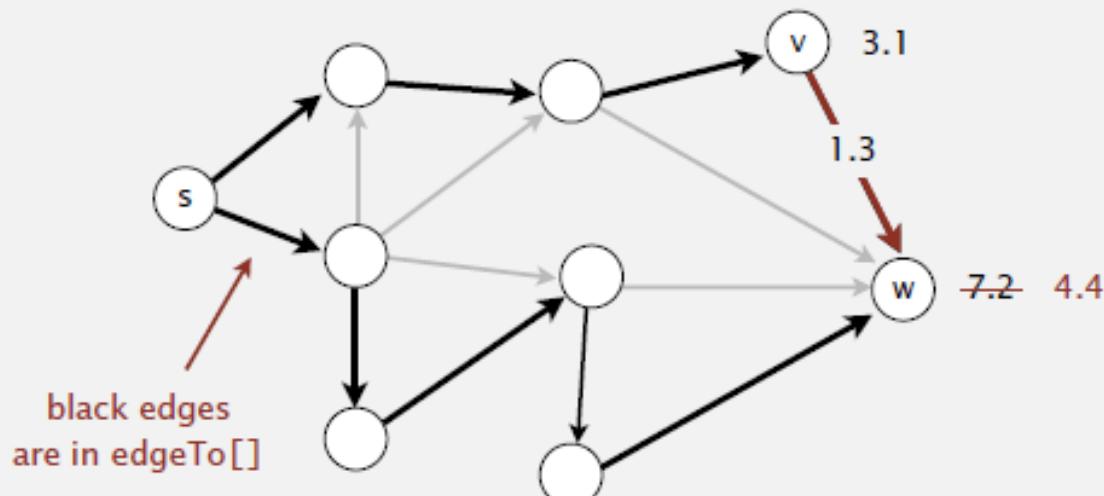
	edgeTo[]	distTo[]
0	null	0
1	5->1	0.32
2	0->2	0.26
3	7->3	0.37
4	0->4	0.38
5	4->5	0.35
6	3->6	0.52
7	2->7	0.34

parent-link representation

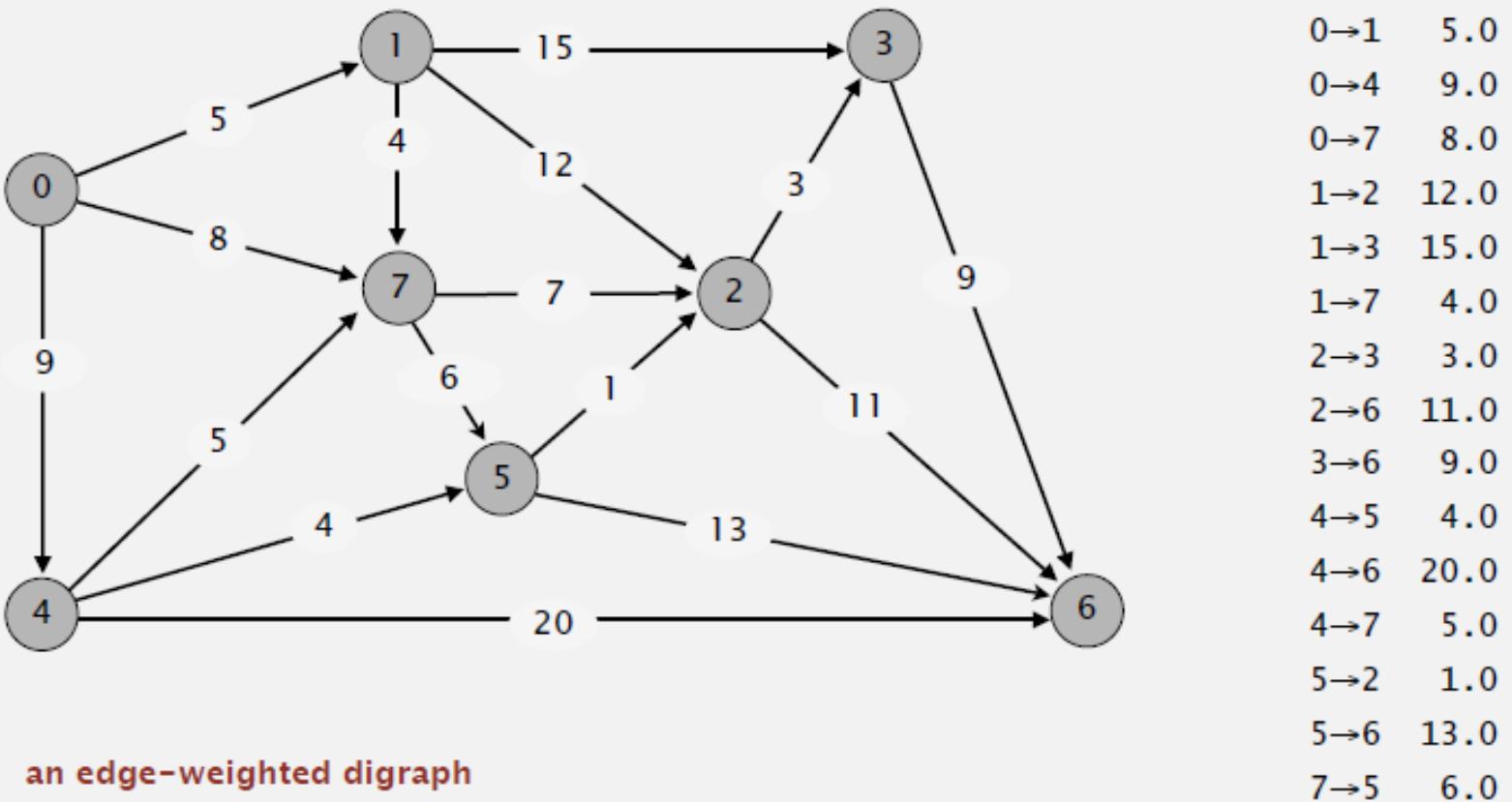
Relax edge $e = v \rightarrow w$.

- $\text{distTo}[v]$ is length of shortest known path from s to v .
- $\text{distTo}[w]$ is length of shortest known path from s to w .
- $\text{edgeTo}[w]$ is last edge on shortest known path from s to w .
- If $e = v \rightarrow w$ gives shorter path to w through v ,
update both $\text{distTo}[w]$ and $\text{edgeTo}[w]$.

$v \rightarrow w$ successfully relaxes



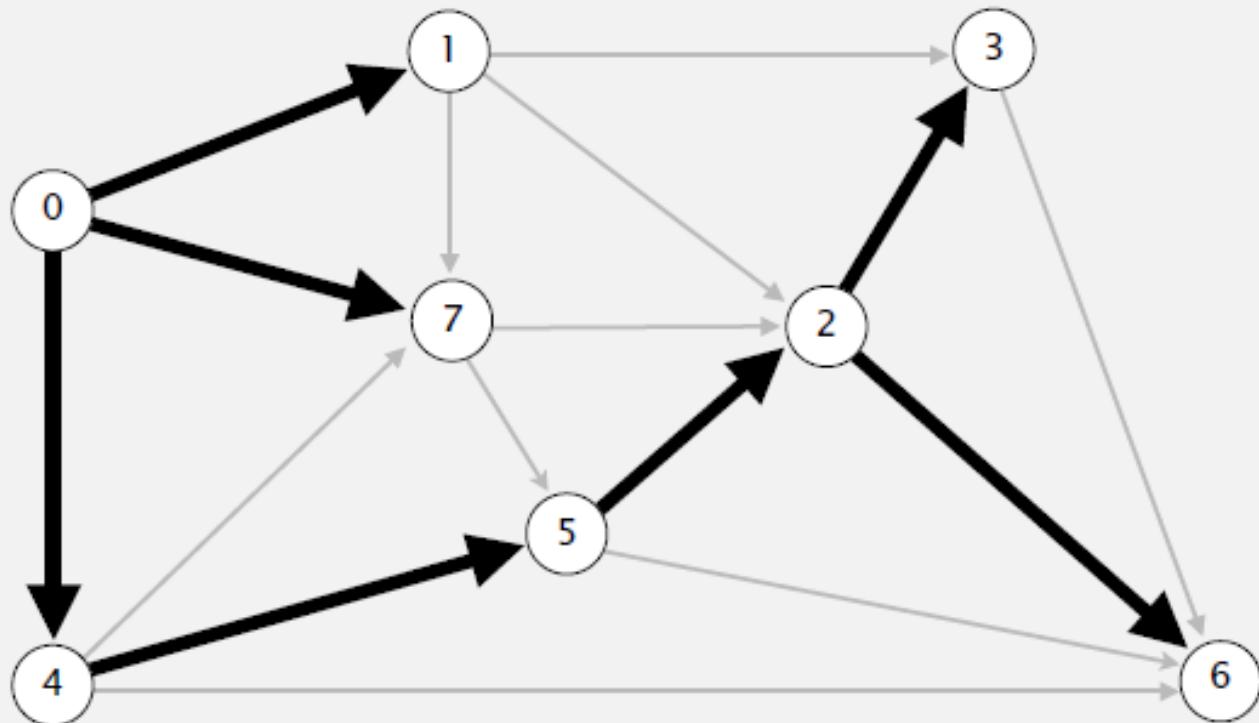
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges pointing from that vertex.



Dijkstra Demo

<https://algs4.cs.princeton.edu/lectures/44DemoDijkstra.pdf>

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



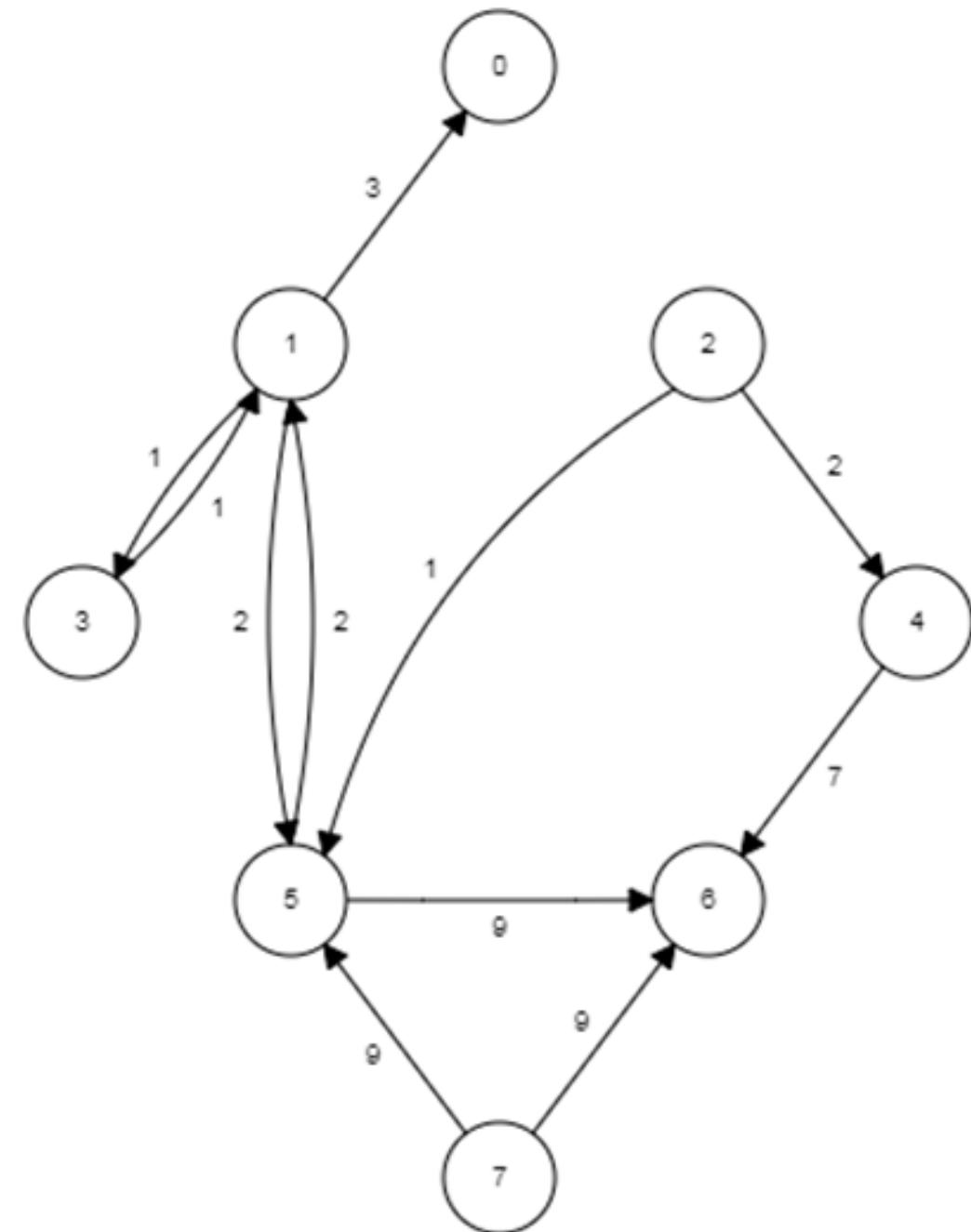
shortest-paths tree from vertex s

v	$\text{distTo}[]$	$\text{edgeTo}[]$
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Dijkstra exercise

Start from vertex 1

v	DistTo[]	EdgeTo[]
0		
1		
2		
3		
4		
5		
6		
7		



Solution

Vertex	Known	Cost	Path	
0	T	3	1	1 0
1	T	0	-1	1
2	F	INF	-1	No Path
3	T	1	1	1 3
4	F	INF	-1	No Path
5	T	2	1	1 5
6	T	11	5	1 5 6
7	F	INF	-1	No Path

Dijkstra performance

Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{dV} V$
Fibonacci heap	1^\dagger	$\log V^\dagger$	1^\dagger	$E + V \log V$

† amortized

Bottom line.

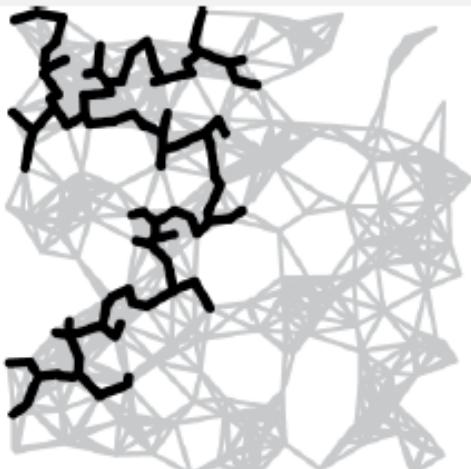
- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

Dijkstra's algorithm seem familiar?

- Prim's algorithm is essentially the same algorithm.
- Both are in a family of algorithms that compute a spanning tree.

Main distinction: Rule used to choose next vertex for the tree.

- Prim: Closest vertex to the **tree** (via an undirected edge).
- Dijkstra: Closest vertex to the **source** (via a directed path).



Dijkstra – why can't work with negative weights?

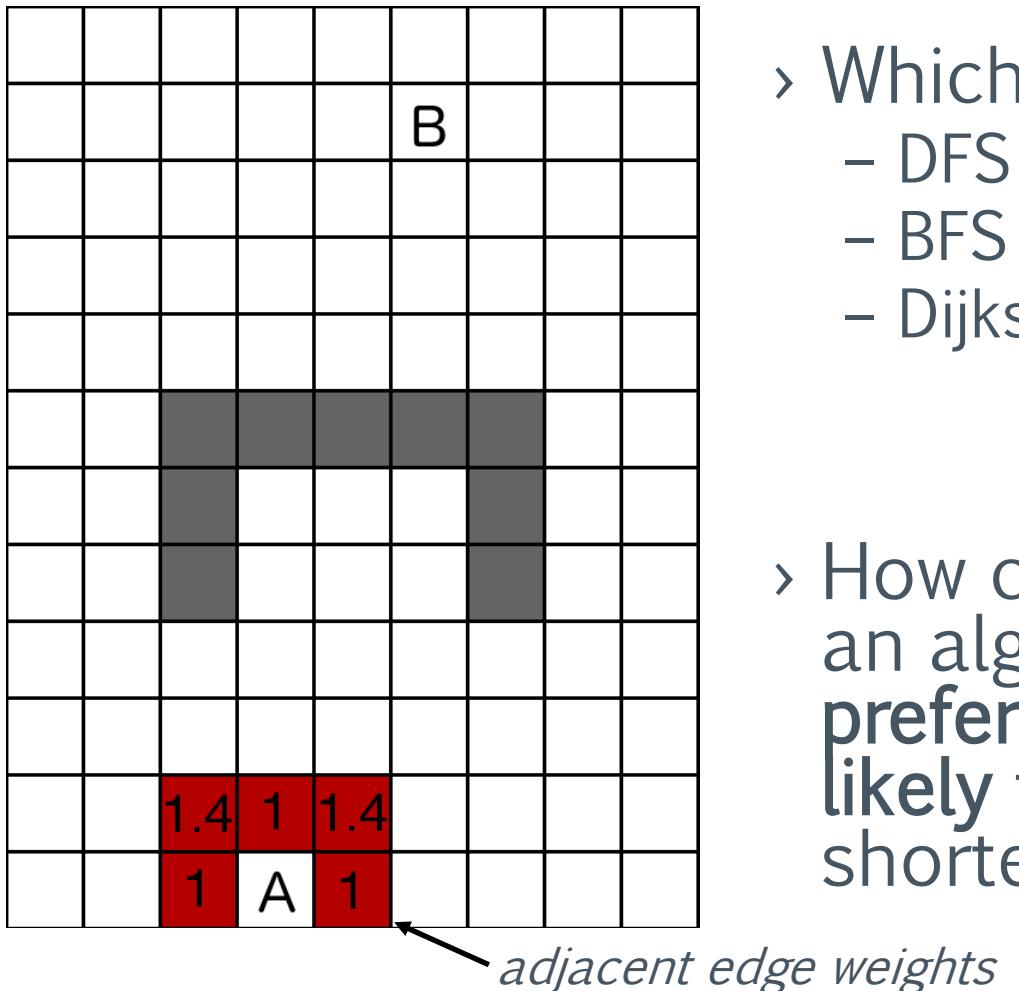
- › Example?
- › Consequence: picking the shortest candidate edge (local optimality) always ends up being correct (global optimality).
 - Greedy algorithm!
 - Wouldn't be true if used dijkstra with negative weights – use a different algorithm instead!
- › Can be modified to work with negative weights, i.e., vertex can be en-queued more than once -> exponential worst case running time though!

Uniform-cost search

- › Algorithm which is focused on finding a single shortest path to a single finishing point rather than the shortest path to every point
- › Stops as soon as the finishing point is found
 - Found a GOAL/SOLUTION in AI
 - Check if at the goal only when selecting the vertex for expansion
- › Summary: expand least-cost-so-far unexpanded vertex

Greedy best-first search

Maze example



- › Which nodes will
 - DFS visit next?
 - BFS visit next?
 - Dijkstra's visit next?
- › How could we write an algorithm that **prefers edges more likely** to be on the shortest path?

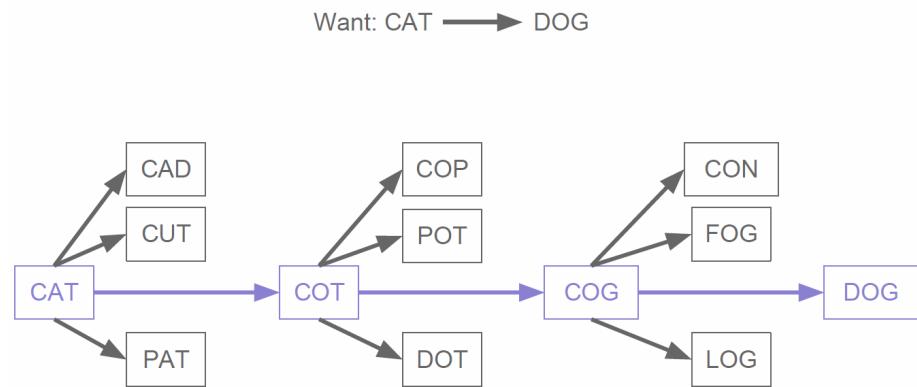
Uninformed vs informed search

- › Dijkstra/uniform cost search - so-called “uninformed” or “blind” search
- › BFS and DFS are blind
 - As likely to go away from the destination than towards it
- › Dijkstra
 - Is looking for shortest paths, in all directions
- › Often we have some knowledge about the destination
 - Domain-specific
 - Informed search
 - What do we do with this knowledge? Heuristic function

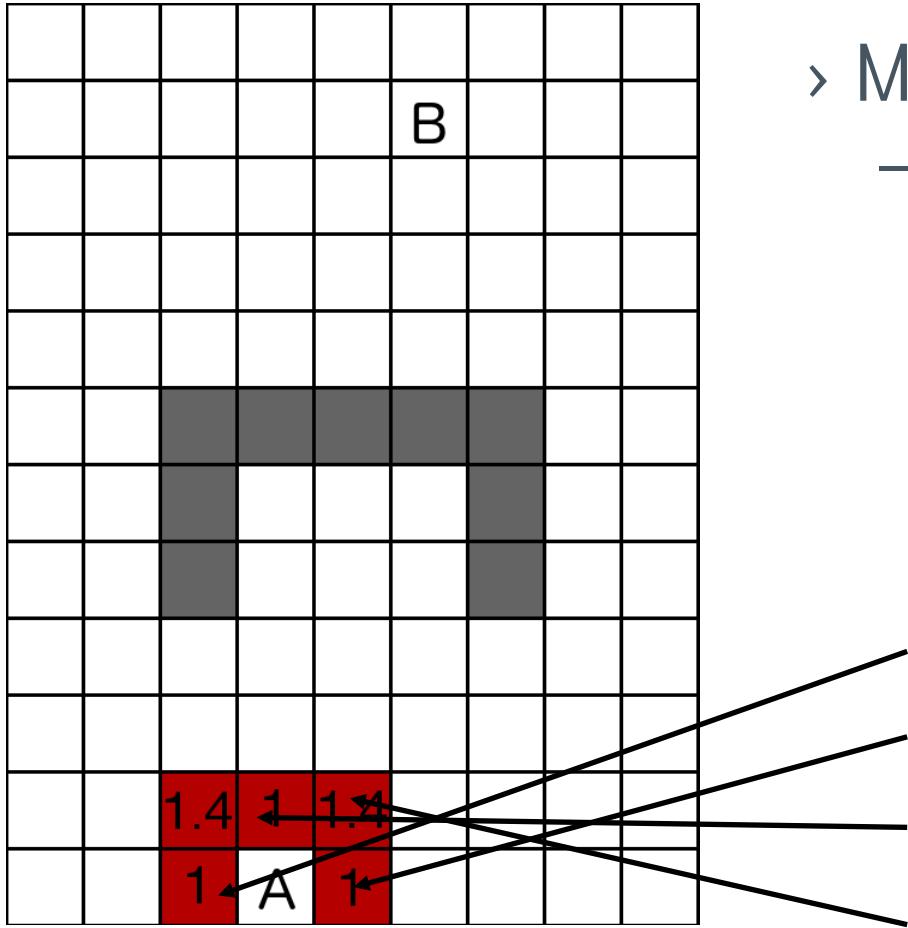
Heuristic

- › Need to estimate how close we are to the destination
 - Use a **heuristic** - an approximate measure of how close you are to the destination/target
 - Should be easy to compute!

- › Examples
 - Google map route planning?
 - › Euclidean distance
 - Mutate the word game
 - › Number of letters correct



Maze example 2



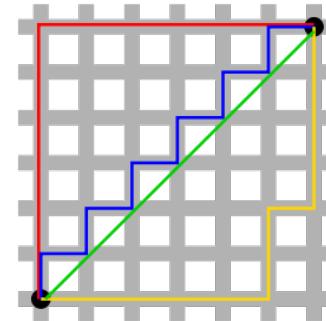
- › Manhattan distance
 - Number of blocs up + number of blocs across

$$10 + 3 = 13$$

$$10 + 1 = 11$$

$$9 + 2 = 11$$

$$9 + 1 = 10$$



Best-first search

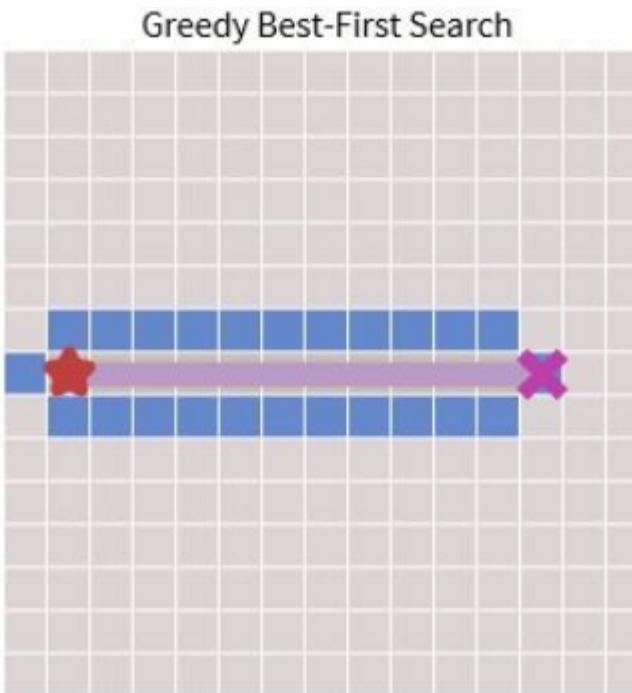
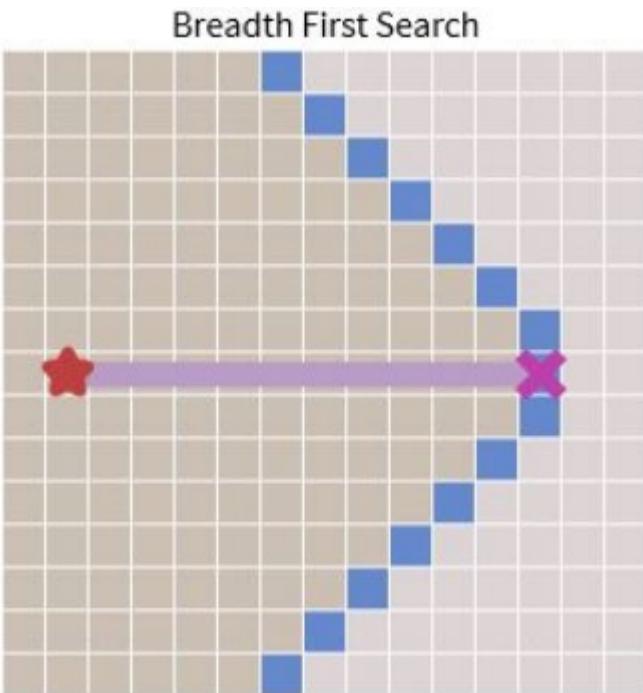
- › Node selected for expansion based on a function $f(n)$
- › $f(n)$ is a cost estimate, and the nodes with the lowest $f(n)$ are expanded first
- › Choice of $f(n)$ = search strategy
- › Special-cases: greedy search and A* search

Greedy best first search

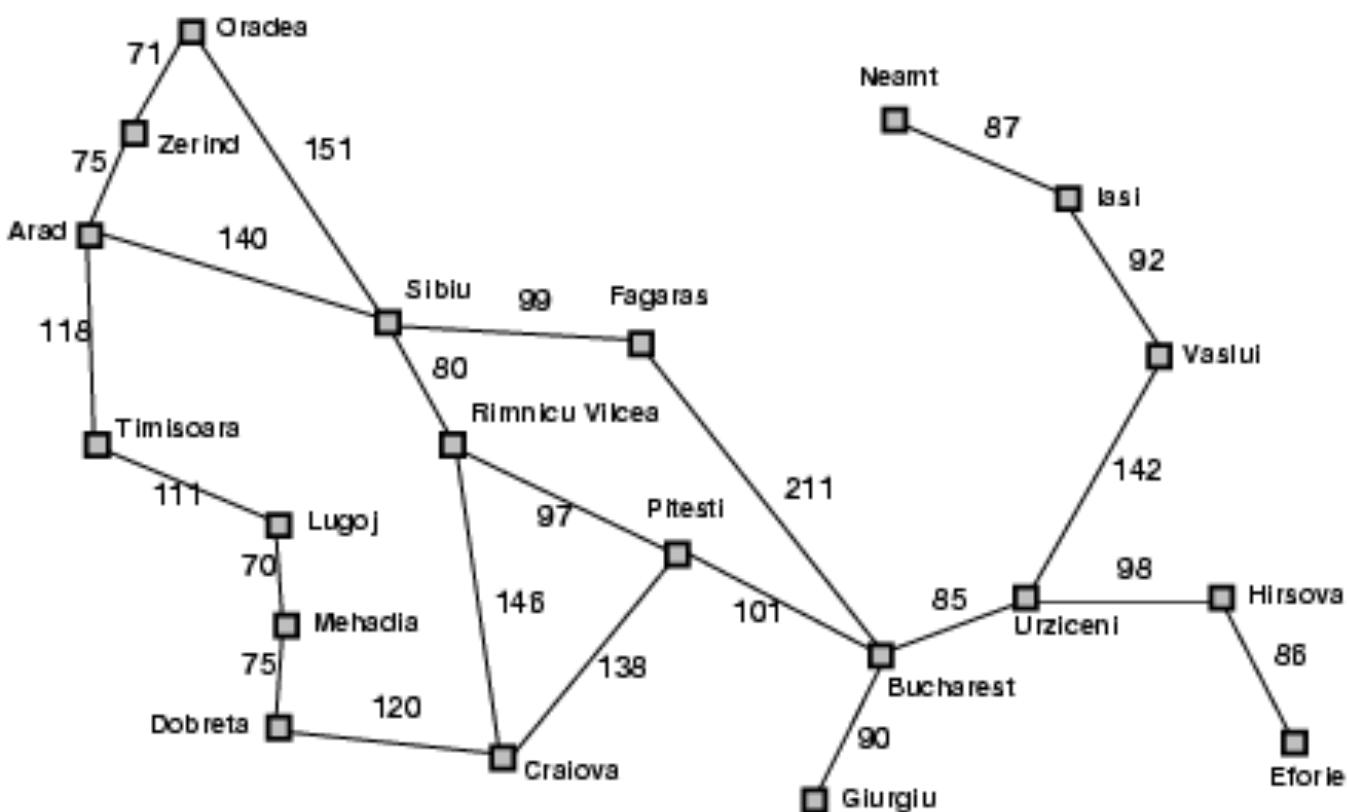
- › Like Breadth-First Search except...
 - queue of choices are ranked using a heuristic
 - › priority queue
 - the parent stays in the queue so that it can back-track
 - stops when goal state found
- › Greedy means choose the **local maximum** at each stage
hoping to find the **global maximum**
 - Dijkstra looks at distance already travelled
 - GBFS looks at distance to go

Greedy best-first search

- › Each step moving closer to the target, e.g.,



Greedy best-first search – Romania example



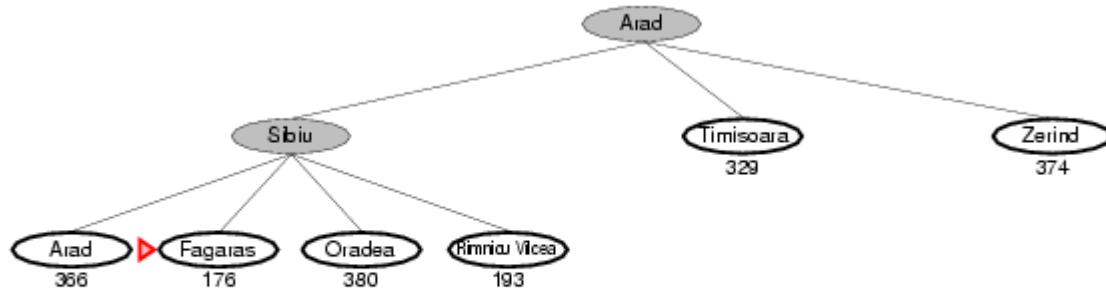
Greedy best-first search example



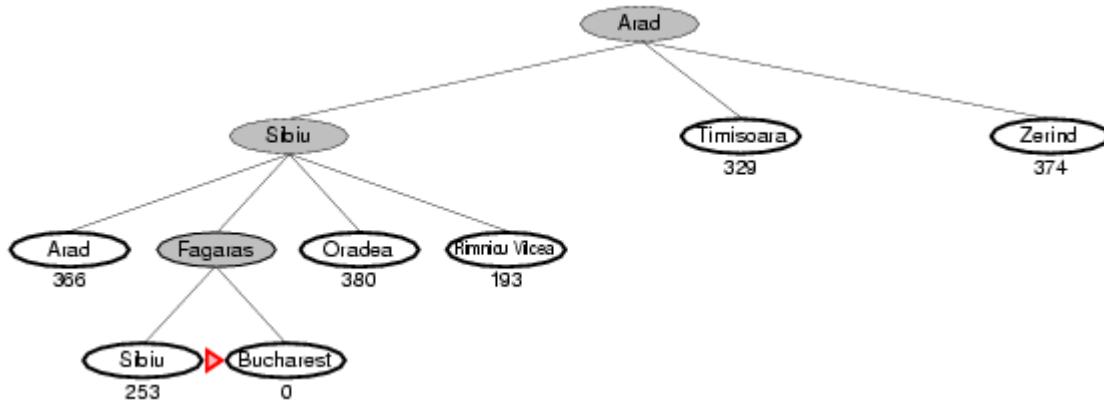
Greedy best-first search example



Greedy best-first search example

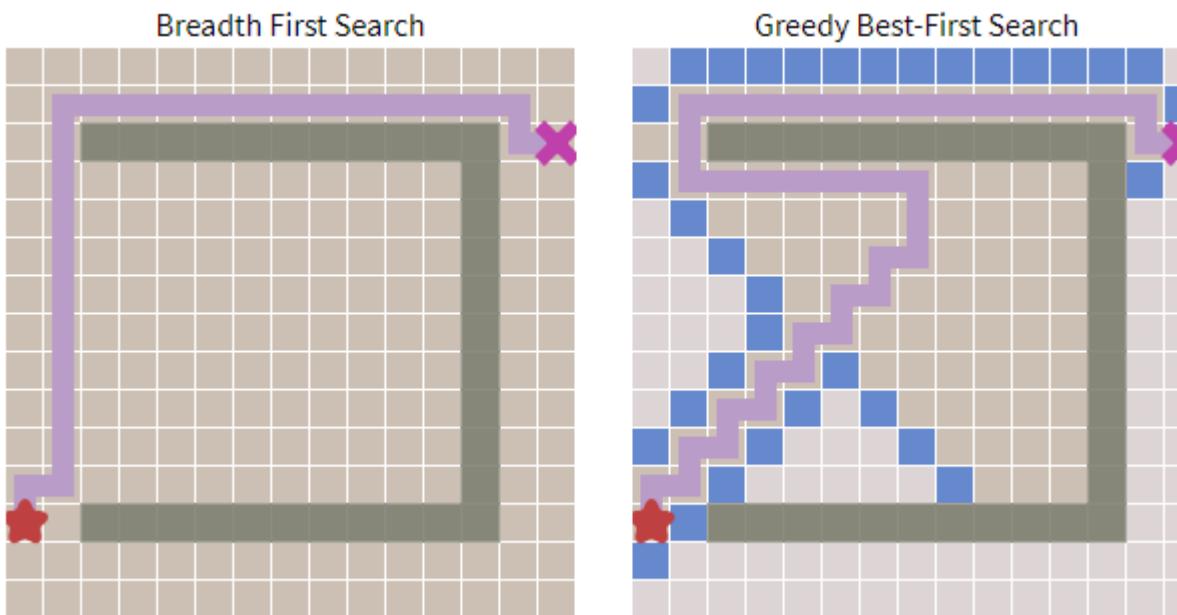


Greedy best-first search example



Optimality

- › Demo of BFS vs Greedy best-first search
- › <https://cs.stanford.edu/people/abisee/tutorial/greedy.html>



Greedy Best First search performance

- › Usually many fewer nodes visited than BFS and Dijkstra
 - Useful when need an answer quickly!
- › Does not guarantee a shortest path like Dijkstra's
 - Greedy = choose what appears best
 - Vulnerable to **local maxima** traps

A*

A* shortest path

- › Single source, single destination/goal
- › Combine Dijkstra's algorithm/Uniform-cost search with greedy best first search
 - Dijkstra looks at minimum cost so far – backward cost
 - Greedy best first looks at minimum estimated cost – forward cost
- › Uses heuristics to guide its search and achieve better performance
 - prioritizes paths that seem to be leading closer to the goal

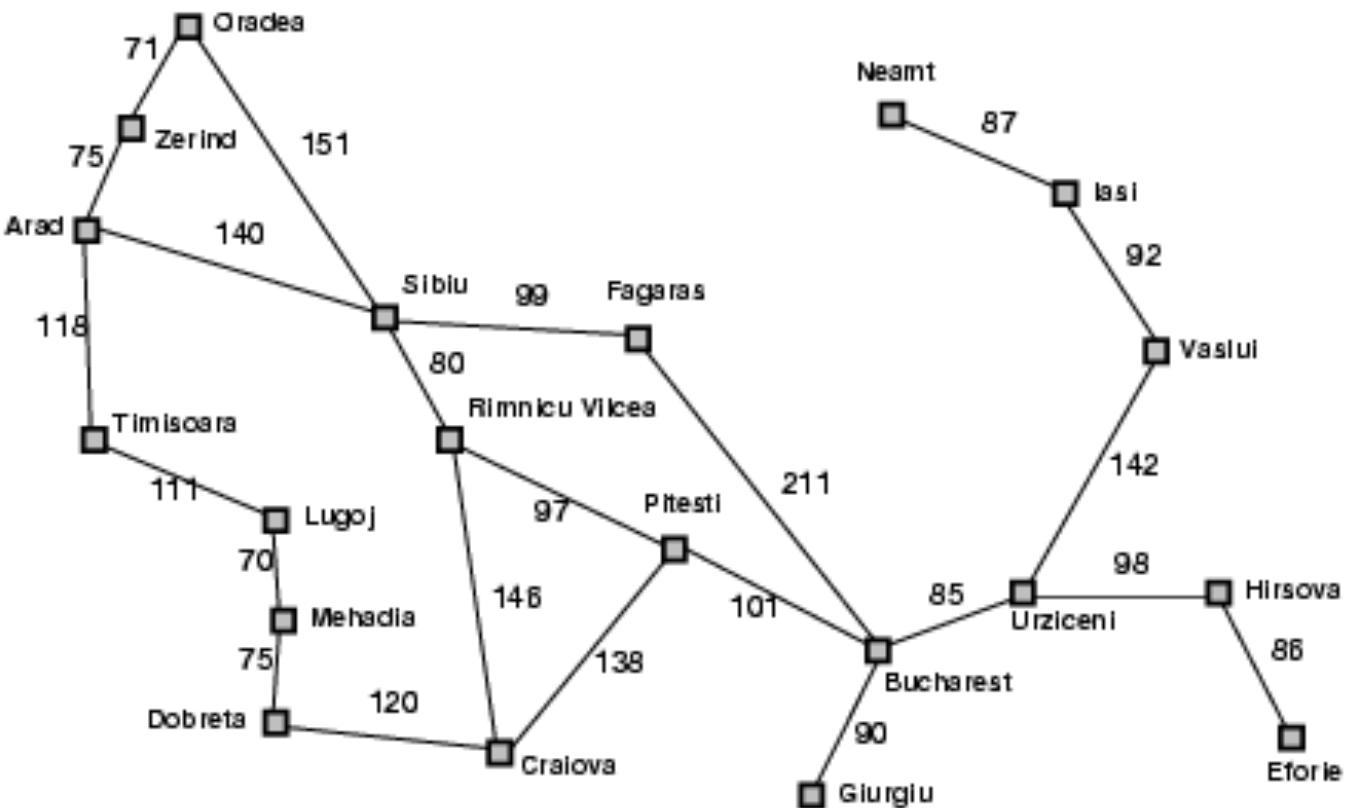
A*

- › actual cost of path so far + estimated cost to goal
 - $f(n) = g(n) + h(n)$
- › This helps avoid local maxima traps
- › May be slower than Greedy Best-First
 - But guarantees shortest path
- › Main idea: avoid paths that have already been expensive

Heuristic function

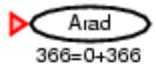
- › solving a problem more quickly when classic methods are too slow
- › finding an approximate solution when classic methods fail to find any exact solution
- › trading optimality, completeness, accuracy, or precision for speed.
- › a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow
 - Eg approximate exact solution

A* – Romania example



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

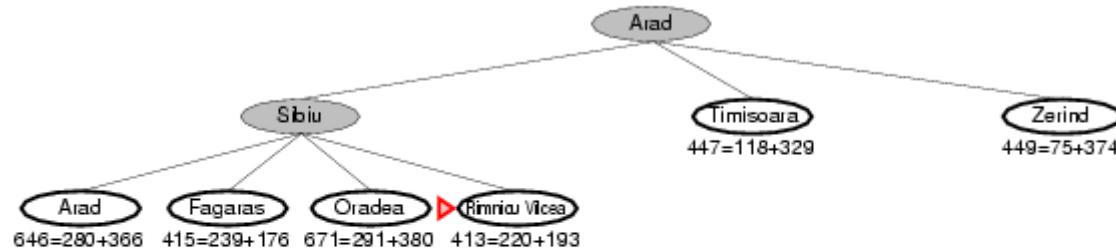
A* search example



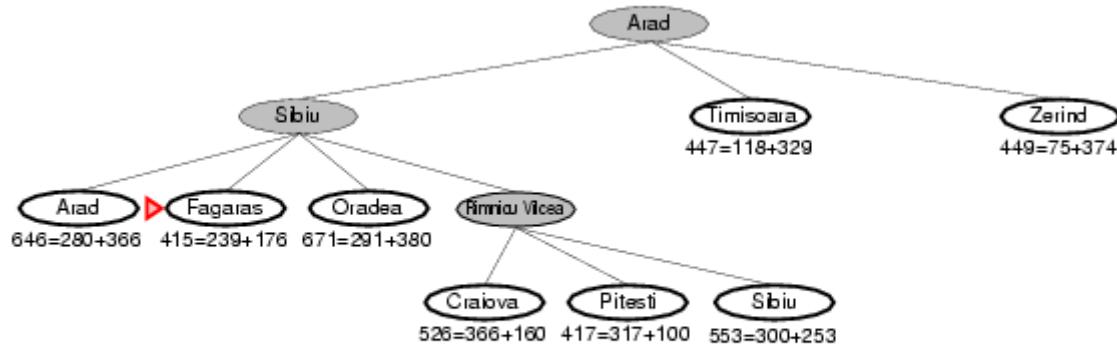
A* search example



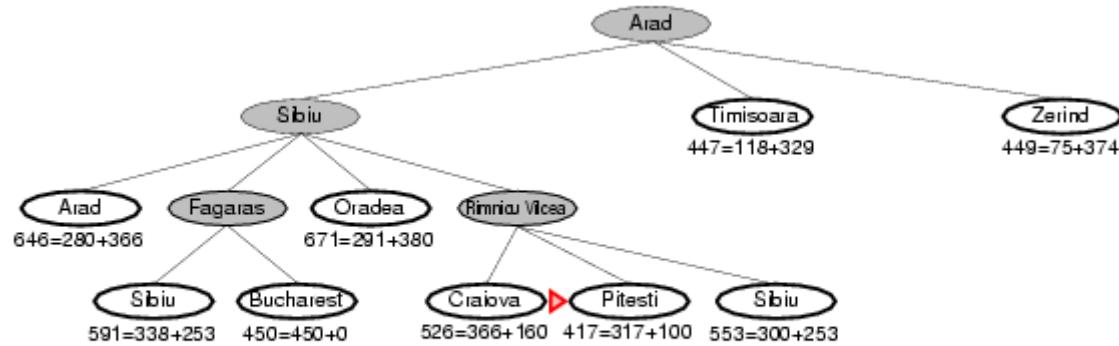
A* search example



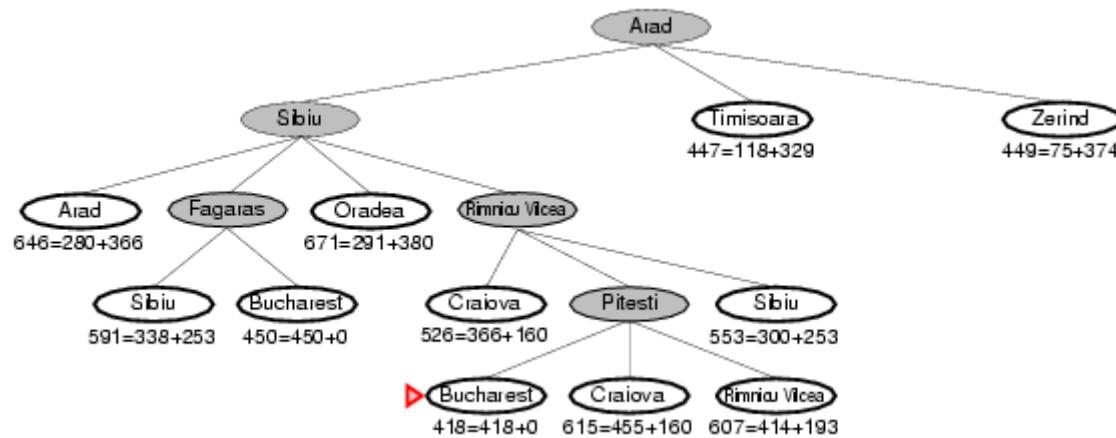
A* search example



A* search example



A* search example



Heuristic function $h(n)$

- › The 2 most important properties:
 - relatively cheap to compute
 - relatively accurate estimator of the cost to reach a goal. Usually a “good” heuristic is if $\frac{1}{2} \text{opt}(n) \leq h(n) \leq \text{opt}(n)$
- › Admissible heuristic

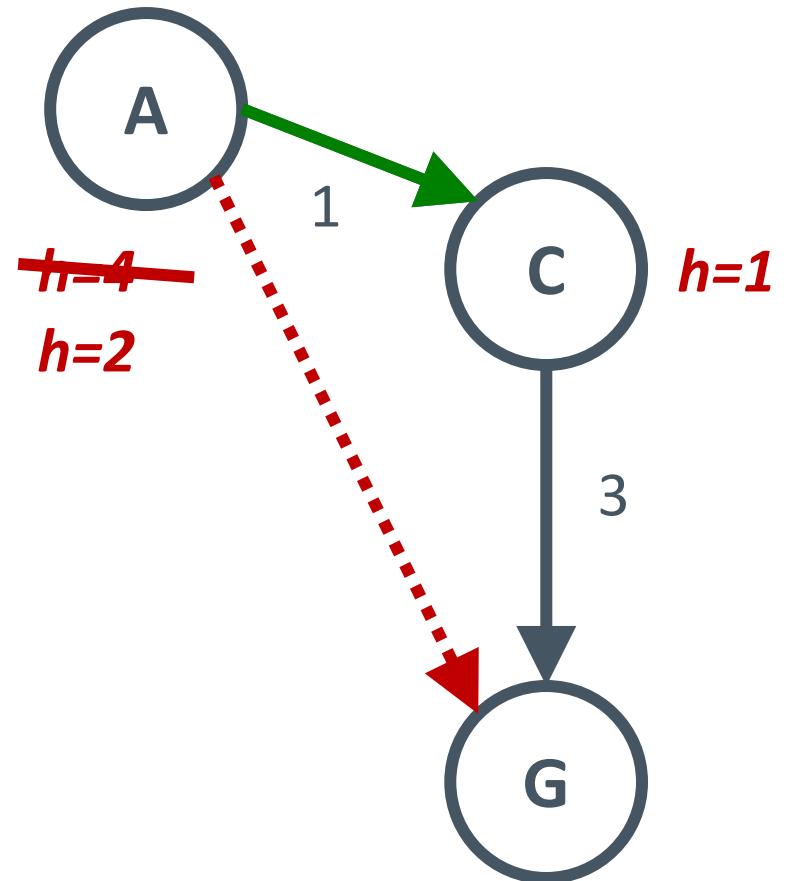
Admissible heuristics

- › A heuristic $h(n)$ is **admissible** if for every node n ,
 $h(n) \leq h^*(n)$, where $h^*(n)$ is the **true** cost to reach the goal state from n .
- › An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is **optimistic**
- › Example: *Romania example straight line distance* - never overestimates the actual road distance

Admissible heuristics

- › Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe

Consistency of Heuristics



- › Main idea: estimated heuristic costs \leq actual costs
 - Admissibility: heuristic cost \leq actual cost to goal
$$h(A) \leq \text{actual cost from } A \text{ to } G$$
 - Consistency: heuristic “arc” cost \leq actual cost for each arc
$$h(A) - h(C) \leq \text{cost}(A \text{ to } C)$$
- › Consequences of consistency:
 - The f value along a path never decreases
$$h(A) \leq \text{cost}(A \text{ to } C) + h(C)$$
 - A* graph search is optimal

A*

- › How to pick a heuristic function?
- › Domain knowledge – some information about a goal
- › Eg in route planning:
 - Manhattan distance – on a square grid that allows 4 directions of movement
 - Diagonal distance – 8-direction square
 - Euclidean distance – any direction
 - A good example on route planning in games
<http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>

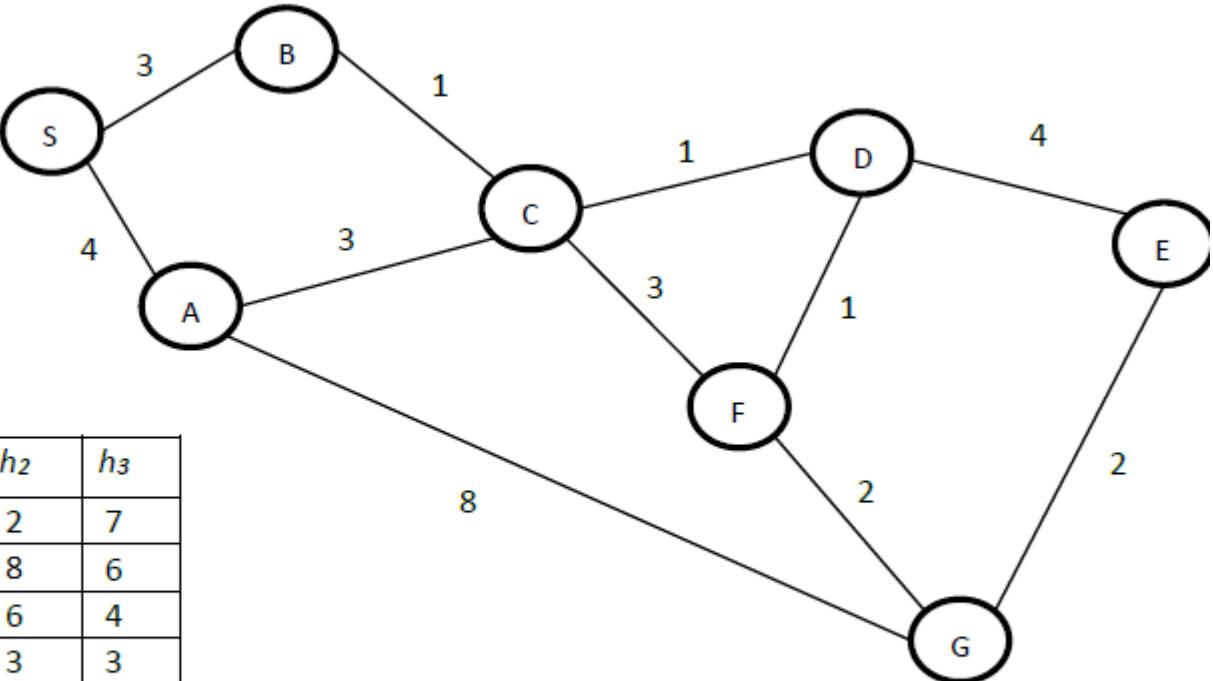
A* vs Dijkstra visualisation

<https://www.youtube.com/watch?v=g024lzsknDo>

A* exercise

- Are heuristics h_1 , h_2 , h_3
 - Admissible?
 - Consistent?
- Find the shortest path from S to G using one of the consistent heuristics

	h_1	h_2	h_3
S	3	2	7
A	6	8	6
B	2	6	4
C	3	3	3
D	2	3	3
E	1	1	1
F	2	2	2
G	0	0	0



Topological sort shortest path

Shortest path in acyclic graphs

- › Is it easier than in graphs with cycles?
- › Yes! Linear time
- › Use topological sort (which only works in DAGs – directional acyclic graphs) to find shortest path
- › If we have an edge from x to y , the ordering visits x before y
- › Shortest path visits/relaxes edges in topological order
- › Topological sort – identify a vertex with no incoming edges, add it to the ordered list, remove it, repeat...
 - (decrease by 1 and conquer)

Topological sort shortest path demo

<https://algs4.cs.princeton.edu/lectures/44DemoAcyclicSP.pdf>

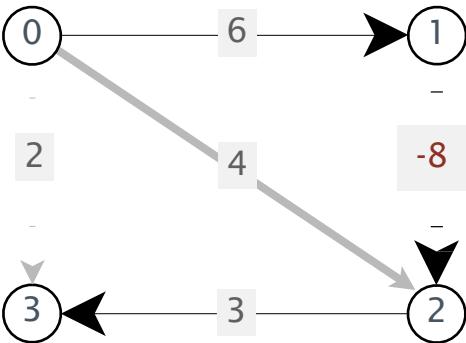
So far

- › No negative weights – Dijkstra
- › No cycles – Topological sort shortest path
- › What if the graph has negative weights and cycles?
- › What about negative cycles?

Bellman-Ford Shortest Path Algorithm

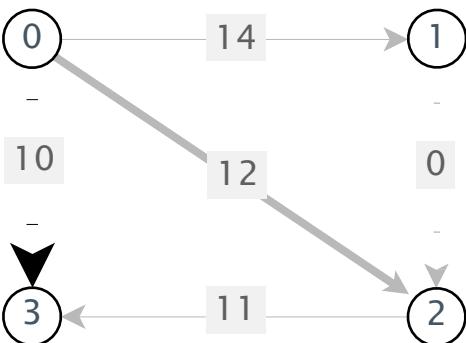
Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0.
But shortest path from 0 to 3 is $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$.

Re-weighting. Add a constant to every edge weight doesn't work.



Adding 8 to each edge weight changes the
shortest path from $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ to $0 \rightarrow 3$.

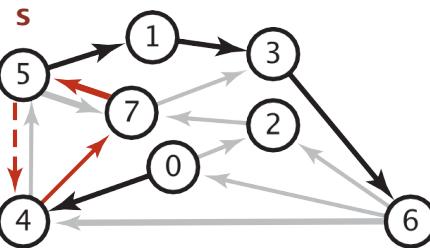
Conclusion. Need a different algorithm.

Negative cycles

Def. A **negative cycle** is a directed cycle whose sum of edge weights is negative.

digraph

4->5	0.35
5->4	-0.66
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



negative cycle (-0.66 + 0.37 + 0.28)

5->4->7->5

shortest path from 0 to 6

0->4->7->5->4->7->5...->1->3->6

Proposition. A SPT exists iff no negative cycles.

assuming all vertices reachable from s

Bellman-Ford algorithm

Bellman-Ford algorithm

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat V times:

- Relax each edge.

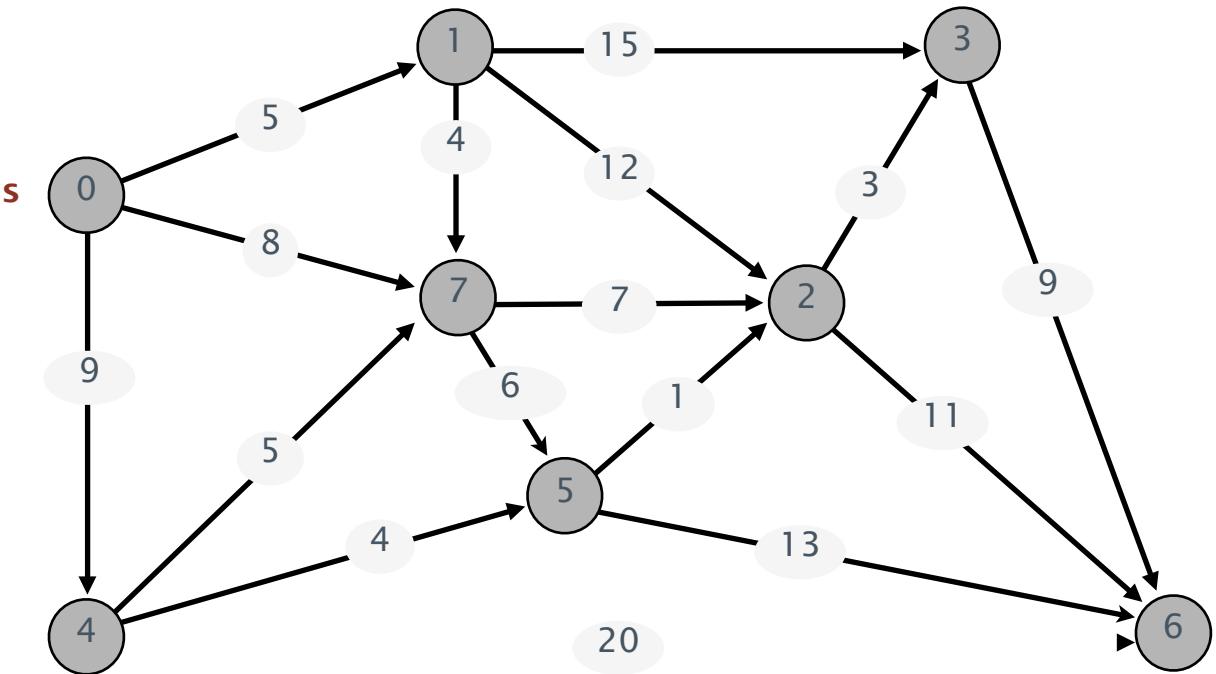
```
for (int i = 0; i < G.V(); i++)
    for (int v = 0; v < G.V(); v++)
        for (DirectedEdge e : G.adj(v))
            relax(e);
```



pass i (relax each edge)

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



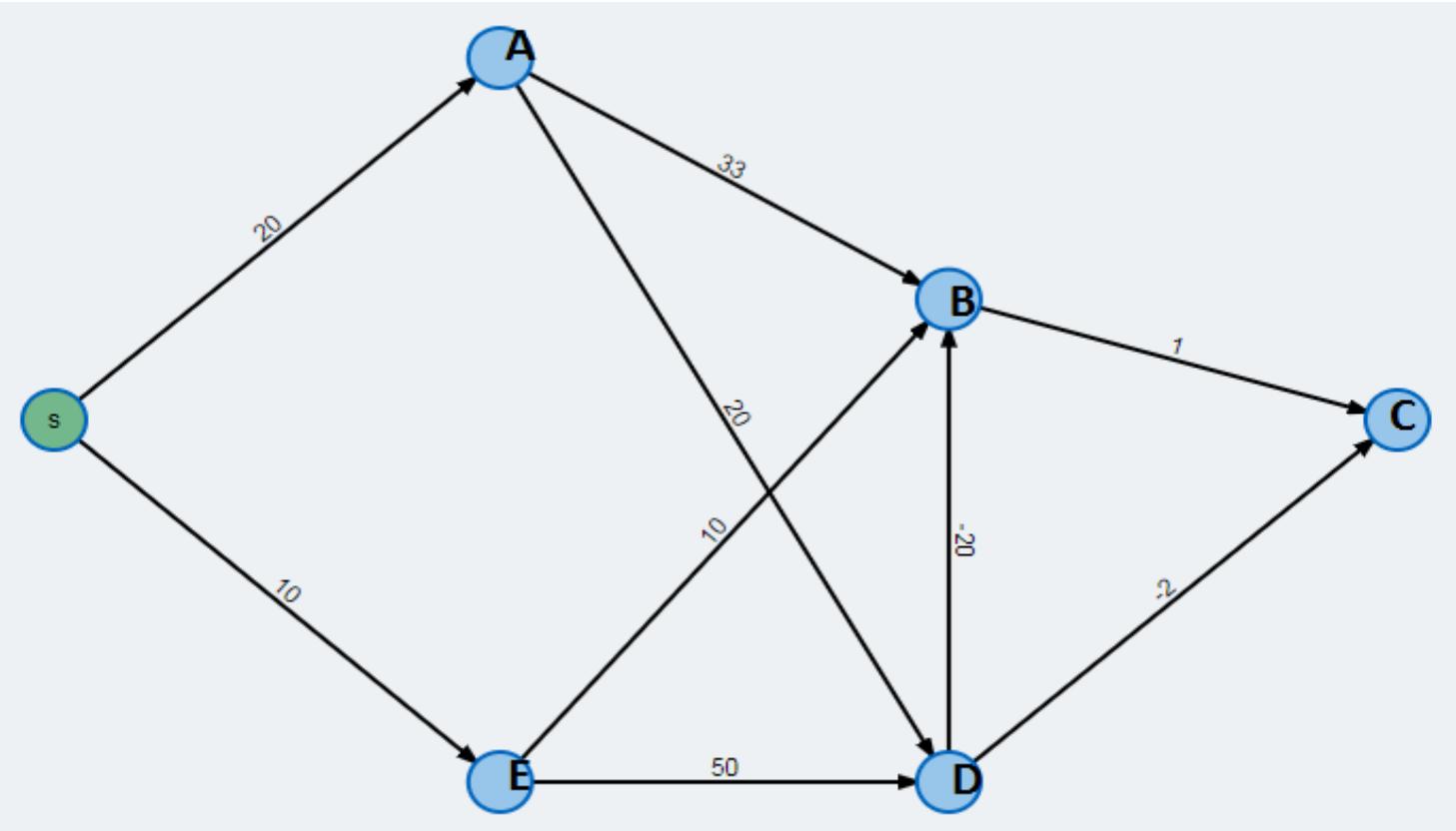
an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
5→7	6.0
6→2	7.0

Bellman-Ford Demo

- › <https://algs4.cs.princeton.edu/lectures/44DemoBellmanFord.pdf>
- › Only positive weights in this example – paper exercise with negative weights

Bellman-Ford Exercise

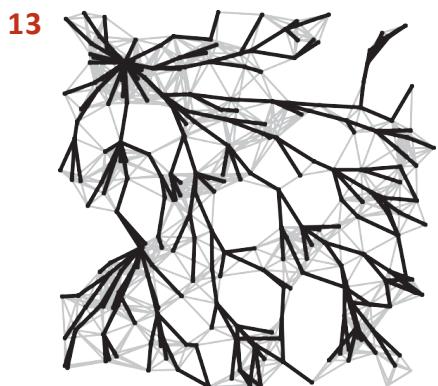
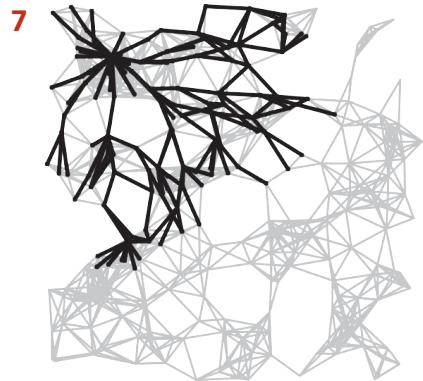
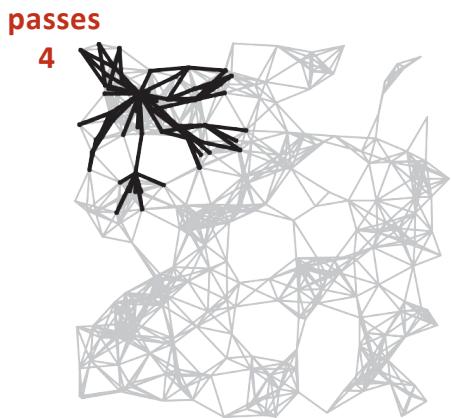


Bellman-Ford Exercise Table

Iteration	Dist to S	A	B	C	D	E
0	0	inf	inf	inf	inf	Inf
1						
2						
3						
4						
5						

Keep track of distance and parent node eg 5 (via B) for each node for each iteration

Bellman-Ford algorithm: visualization



Bellman-Ford algorithm: analysis

Bellman-Ford algorithm

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat V times:

- Relax each edge.
-

Proposition. Dynamic programming algorithm computes SPT in any edge-weighted digraph with no negative cycles in time proportional to $E \times V$.

Pf idea. After pass i , found path that is at least as short as any shortest path containing i (or fewer) edges.

Bellman-Ford algorithm: practical improvement

Observation. If $\text{distTo}[v]$ does not change during pass i , no need to relax any edge pointing from v in pass $i+1$.

FIFO implementation. Maintain **queue** of vertices whose $\text{distTo}[]$ changed.



be careful to keep at most one copy
of each vertex on queue (why?)

Overall effect.

- The running time is still proportional to $E \times V$ in worst case.
- But much faster than that in practice.

Single source shortest path summary

algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	$E + V$	$E + V$	V
Dijkstra (binary heap)	no negative weights	$E \log V$	$E \log V$	V
Bellman-Ford	no negative cycles	EV	EV	V
Bellman-Ford (queue-based)		$E + V$	EV	V

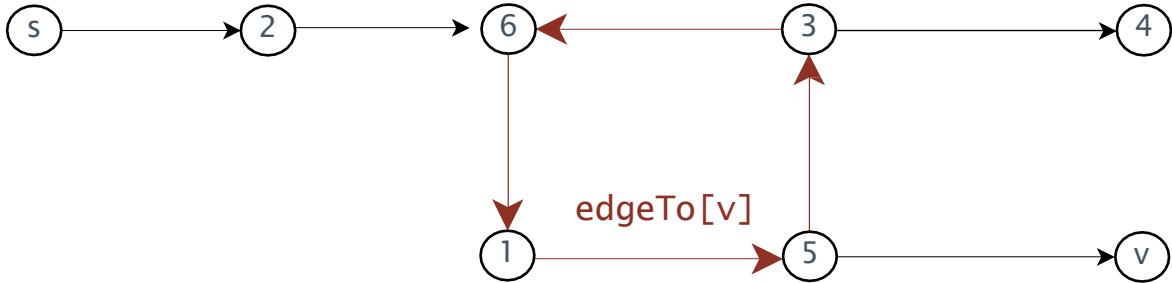
Remark 1. Directed cycles make the problem harder.

Remark 2. Negative weights make the problem harder.

Remark 3. Negative cycles makes the problem intractable.

Finding a negative cycle

Observation. If there is a negative cycle, Bellman-Ford gets stuck in loop, updating `distTo[]` and `edgeTo[]` entries of vertices in the cycle.



Proposition. If any vertex v is updated in pass v , there exists a negative cycle (and can trace back `edgeTo[v]` entries to find it).

In practice. Check for negative cycles more frequently.

Negative cycle application: arbitrage detection

Problem. Given table of exchange rates, is there an arbitrage opportunity?

	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.011
EUR	1.350	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.943	0.698	0.620	1	0.953
CAD	0.995	0.732	0.650	1.049	1

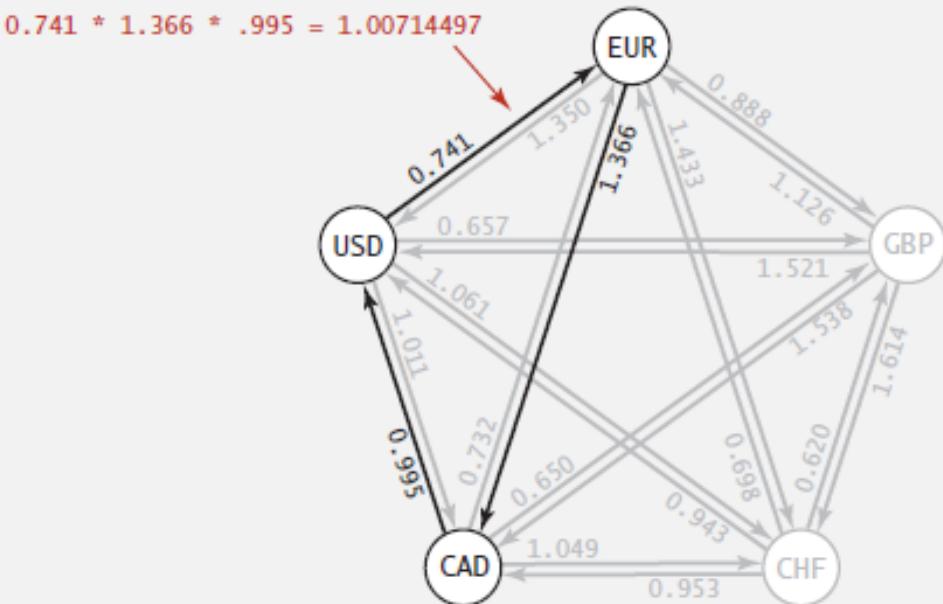
Ex. \$1,000 \Rightarrow 741 Euros \Rightarrow 1,012.206 Canadian dollars \Rightarrow \$1,007.14497.

$$1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497$$

Negative cycle application: arbitrage detection

Currency exchange graph.

- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find a directed cycle whose product of edge weights is > 1 .

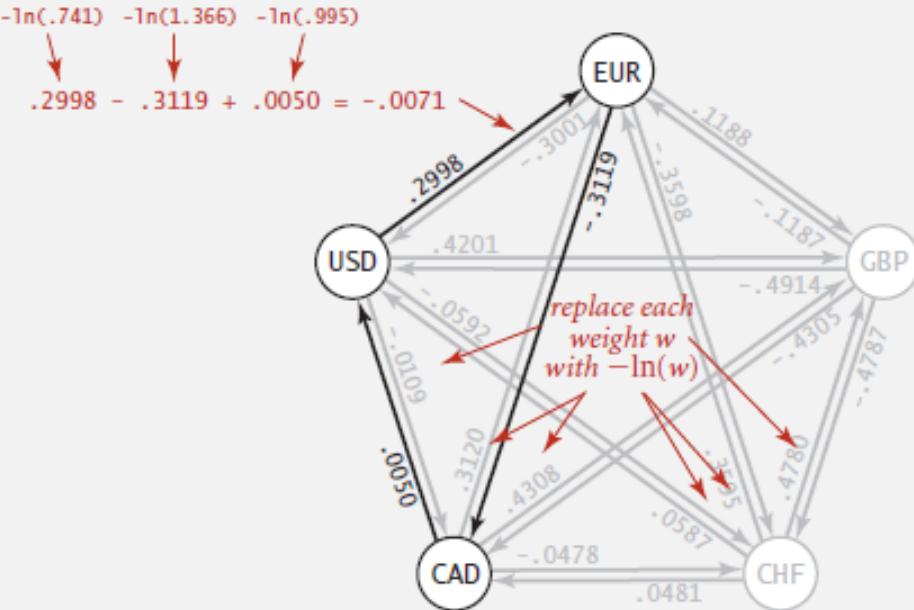


Challenge. Express as a negative cycle detection problem.

Negative cycle application: arbitrage detection

Model as a negative cycle detection problem by taking logs.

- Let weight of edge $v \rightarrow w$ be $-\ln$ (exchange rate from currency v to w).
- Multiplication turns to addition; > 1 turns to < 0 .
- Find a directed cycle whose sum of edge weights is < 0 (negative cycle).



Remark. Fastest algorithm is extraordinarily valuable!

Floyd-Warshall shortest path algorithm

All-pairs shortest path

- › Run Dijkstra for each vertex?
- › Run Bellman-Ford each vertex?
- › Worst case scenario – assume fully connected graph, so
 $E = V^2$
- › Dijkstra = V times $V^2 \log V = V^3 \log V$
- › Bellman-Ford = V times $V^3 = V^4$
- › Other options?
- › Floyd-Warshall = V^3 (but V^2 space)

Floyd-Warshall all-pairs shortest path

- › For a path $p = \{v_1, v_2, \dots, v_l\}$, vertices v_2 to v_{l-1} are intermediate vertices
- › Path consisting of a single edge has no intermediate vertices
- › $d_{ij}^{(k)}$ is a shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, 2, \dots, k\}$ – in any order, any subset of them

Floyd-Warshall

- › Assume we know $d_{ij}^{(k-1)}$ – how do we calculate $d_{ij}^{(k)}$
- › The path either goes through k, or it does not
 - If it does not, then shortest path $d_{ij}^{(k)} = d_{ij}^{(k-1)}$
 - If it does, it means it passes through it exactly once, and it consists of shortest path from i to k, and then the shortest path from k to j $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

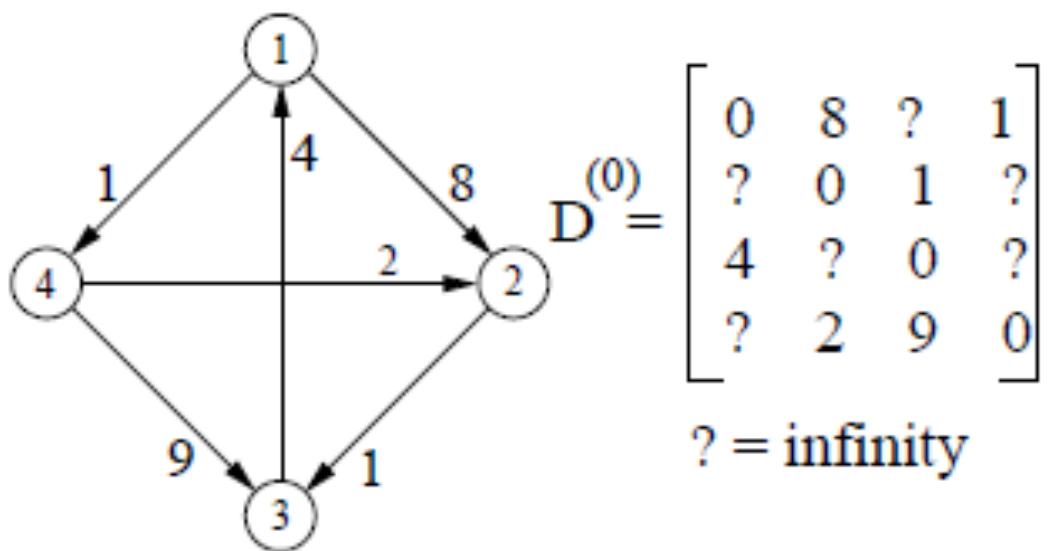
$$d_{ij}^{(0)} = w_{ij},$$

$$d_{ij}^{(k)} = \min \left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) \quad \text{for } k \geq 1.$$

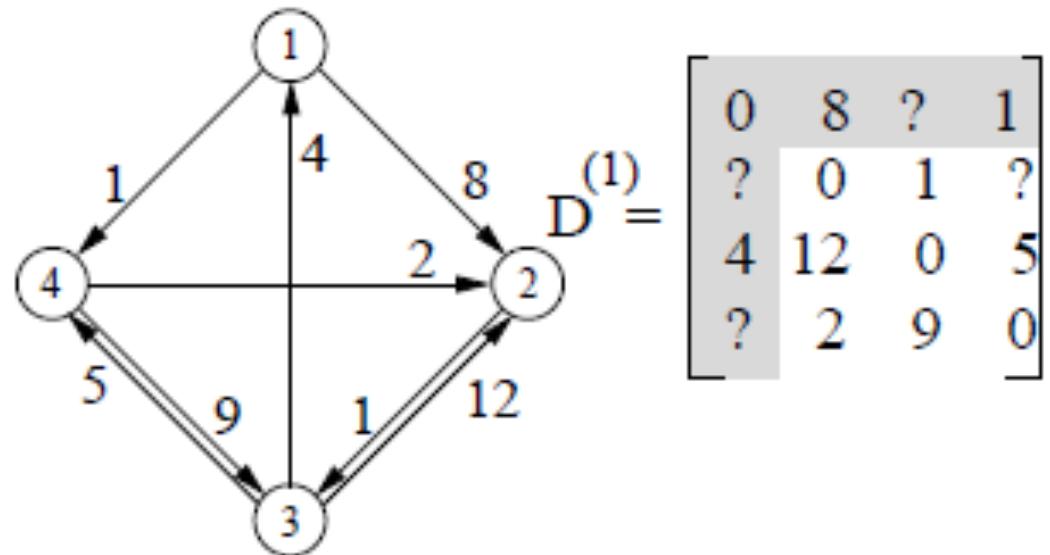
Floyd-Warshall pseudocode

```
for  $k = 1$  to  $n$  do
    for  $i = 1$  to  $n$  do
        for  $j = 1$  to  $n$  do
            if ( $d[i, k] + d[k, j] < d[i, j]$ )
                { $d[i, j] = d[i, k] + d[k, j];$ 
                  $pred[i, j] = k;$ }
    return  $d[1..n, 1..n];$ 
```

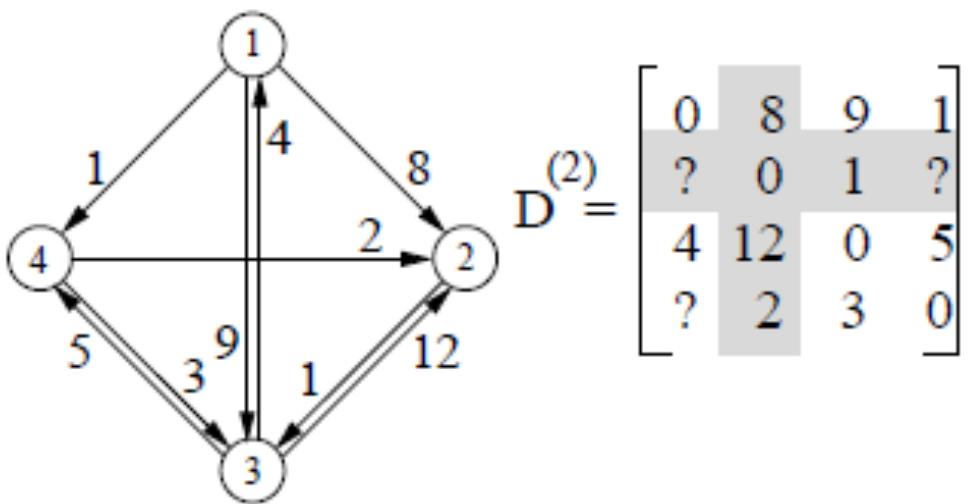
Floyd Warshall example



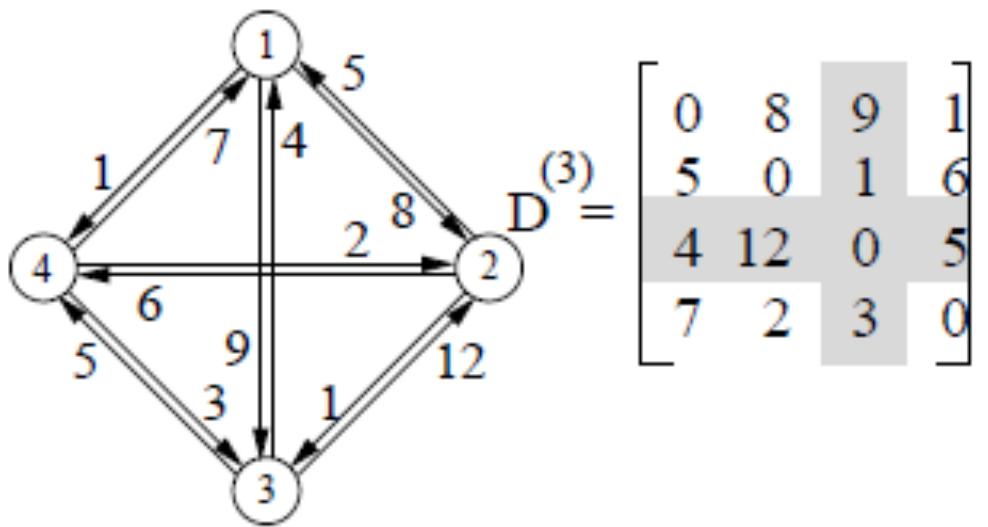
Floyd Warshall example



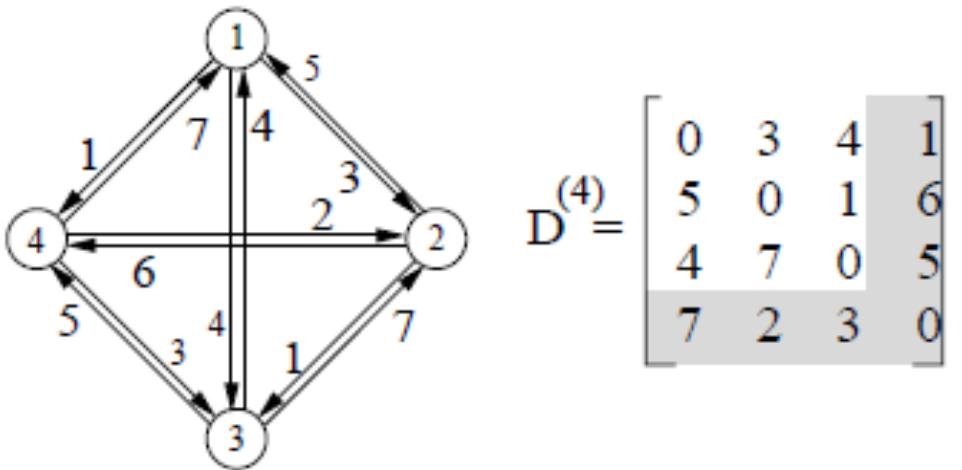
Floyd Warshall example



Floyd Warshall example



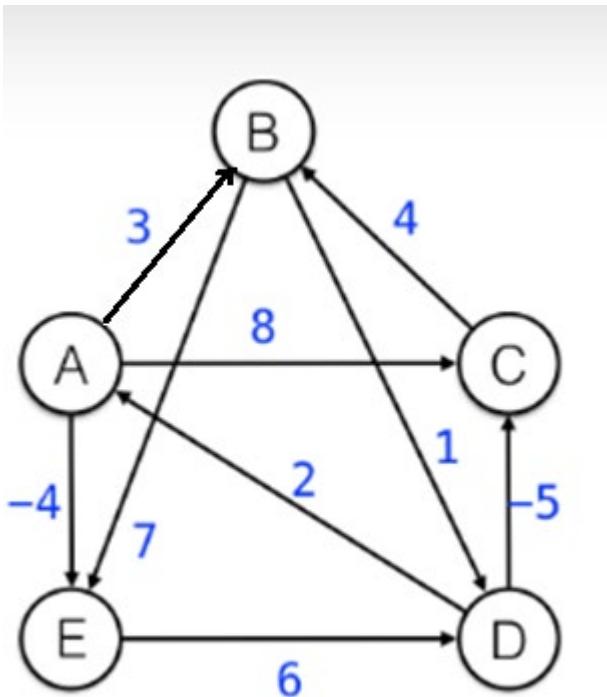
Floyd Warshall example



Floyd-Warshall demo

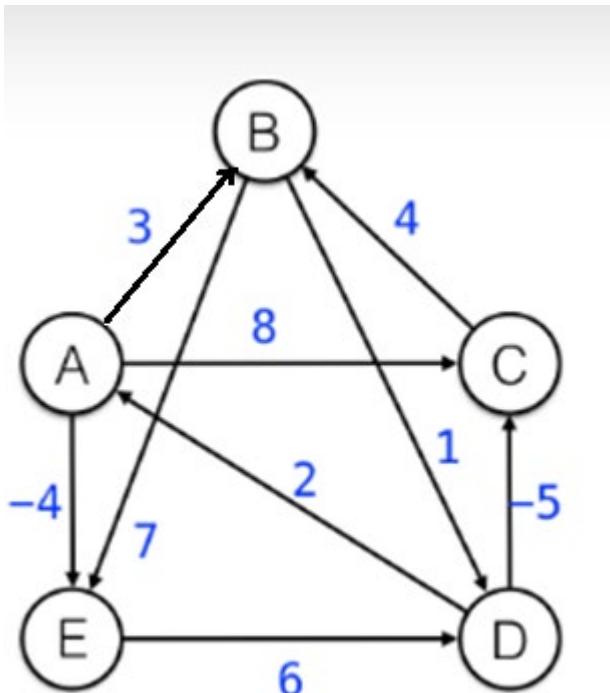
› <https://www.cs.usfca.edu/~galles/visualization/Floyd.html>

Floyd-Warshall exercise



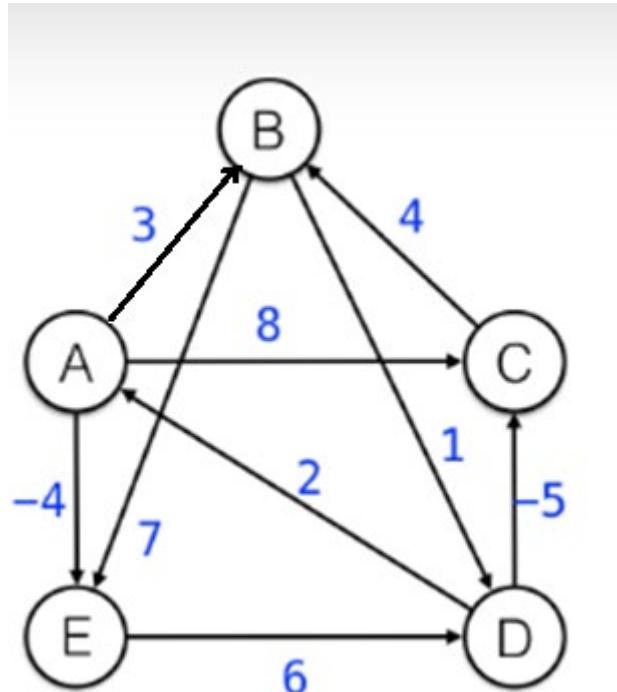
Do	A	B	C	D	E
A					

Floyd-Warshall exercise



D^0	A	B	C	D	E
A	0	3	8	INF	-4
B	INF	0	INF	1	7
C	INF	4	0	INF	INF
D	2	INF	-5	0	INF
E	INF	INF	INF	6	0

Exercise solution



D ⁵	A	B	C	D	E
A	0	0	1	-3	-4
B	3	0	-4	1	-1
C	7	4	0	5	3
D	2	-1	-5	0	-2
E	8	5	1	6	0

Dynamic Programming

Dynamic programming

- › Algorithm Design, Kleinberg and Tardos, Pearson 2014
- › Introduction to Design and Analysis of Algorithms, Levitin. Pearson 2012

Dynamic programming

- › Examples:
 - Bellman-Ford
 - Floyd-Warshall
- › Examines the full search space but implicitly, by breaking up the problem into a series of subproblems, and then building up the solution to larger and larger subproblems
- › Typically overlapping subproblems – instead of over and over calculating solutions to a subproblem, record it in a table and look up when needed
- › So why a fancy name if this is all that it's doing?
- › Richard Bellman on the Birth of Dynamic Programming, by Stuart Dreyfus
<https://pubsonline.informs.org/doi/pdf/10.1287/opre.50.1.48.17791>

Dynamic programming

- › Informal guidelines:

- There are only a polynomial number of subproblems
- The solution to the original problem can be easily computed from the solution of the subproblems
- There is a natural order of subproblems from smallest to largest together with easy to compute recurrence that allows building a solution to a subproblem from smaller subproblems

Shortest paths challenges

- › <http://www.diag.uniroma1.it/challenge9/>
- › <http://www.diag.uniroma1.it/challenge9/download.shtml>

CS2010: Data Structures and Algorithms II

Tries

Ivana.Dusparic@scss.tcd.ie

Where are we?

- › Sorting algorithms
 - General
 - String-specific
- › Exact pattern matching algorithms/substring search
- › Symbol tables – ordered/unordered
 - Lists, queues, hashtables
 - Trees – binary search tree, 2-3 tree, red-black BST
 - Can we do better with string-specific symbol tables?

Binary search tree (BST)

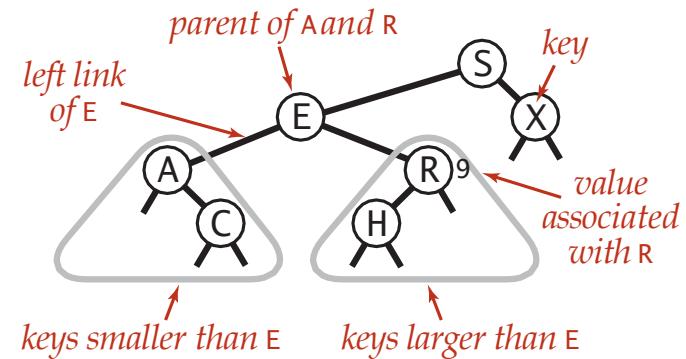
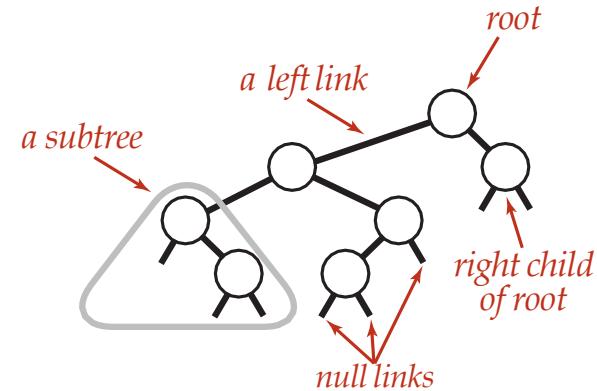
Definition. A BST is a **binary tree in symmetric order**.

A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right).

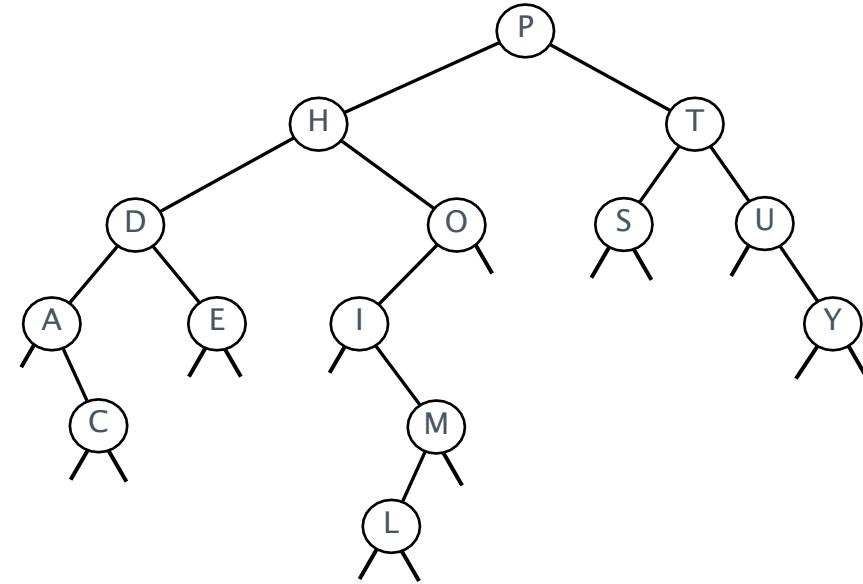
Symmetric order. Each node has a key, and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



Correspondence between BSTs and quicksort partitioning

0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
H	L	E	A	D	O	M	C	I	P	T	Y	U	S
D	C	E	A	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y

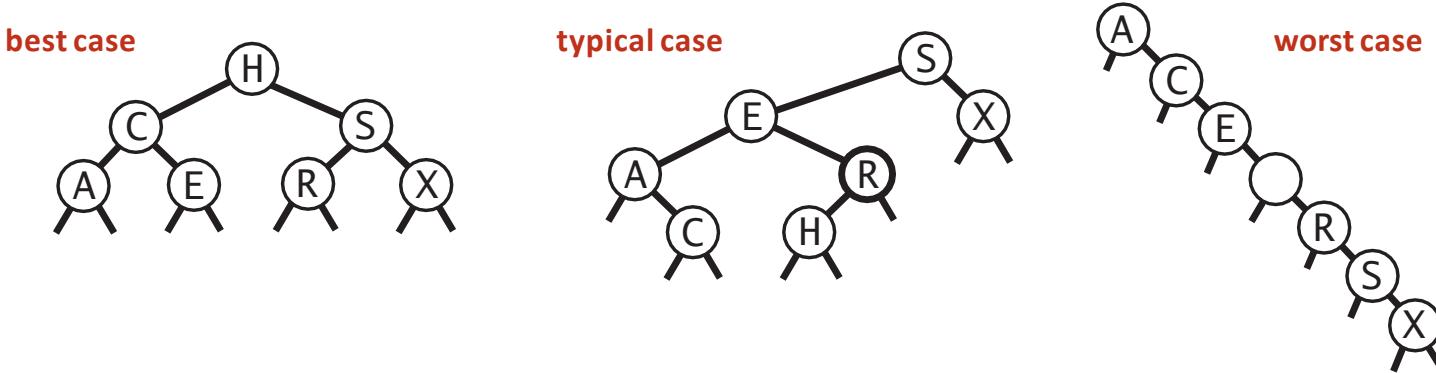


Remark. Correspondence is 1–1 if array has no duplicate keys.

Property. Inorder traversal of a BST yields keys in ascending order.

Tree shape

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to $1 + \text{depth of node}$.



Bottom line. Tree shape depends on order of insertion.

Balanced Search Trees

- › 2-3 tree
- › Red-black BST – BST implementation of 2-3 tree

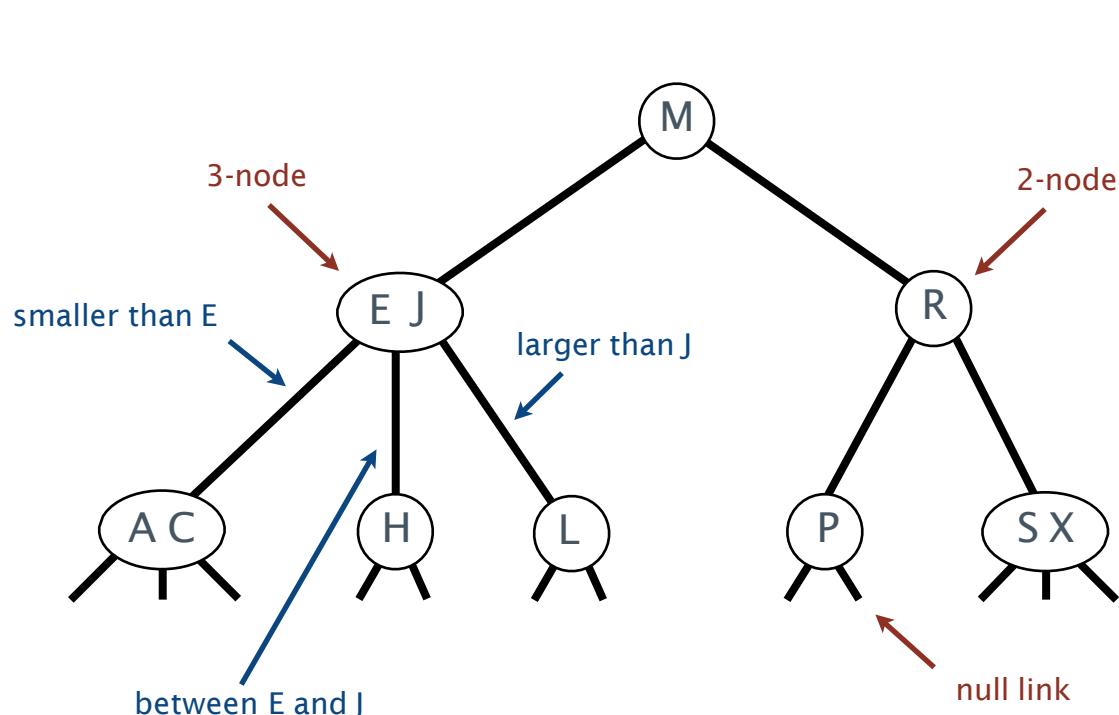
2-3 tree

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

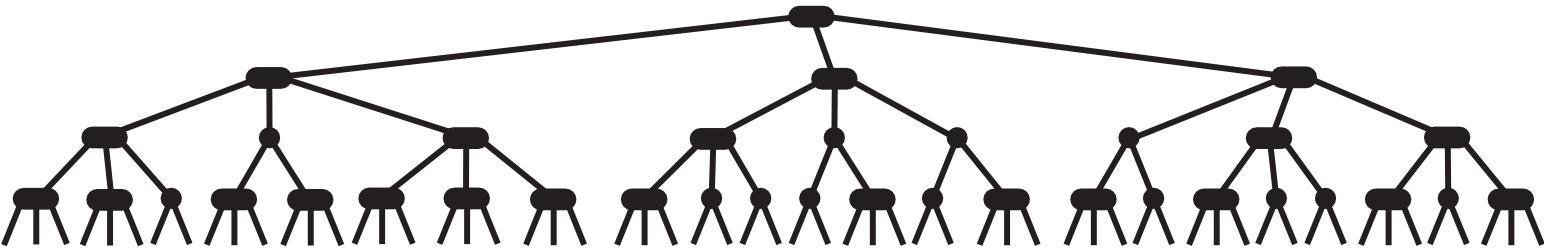
Symmetric order. Inorder traversal yields keys in ascending order.

Perfect balance. Every path from root to null link has same length.



2-3 tree: performance

Perfect balance. Every path from root to null link has same length.



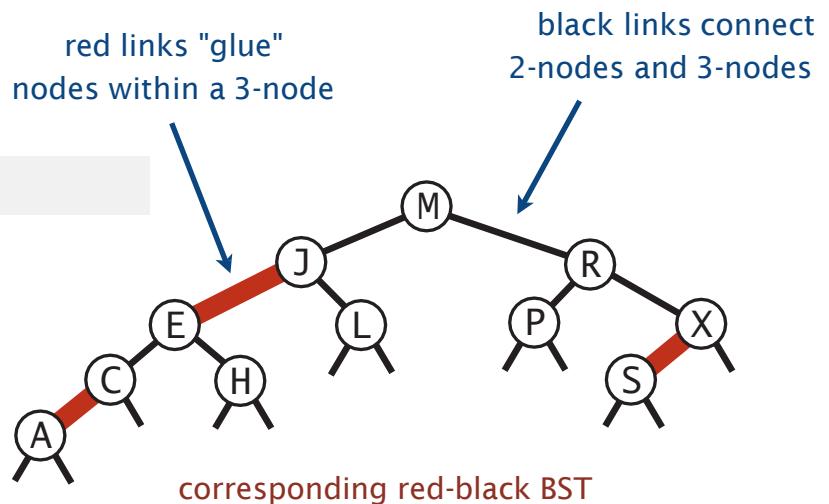
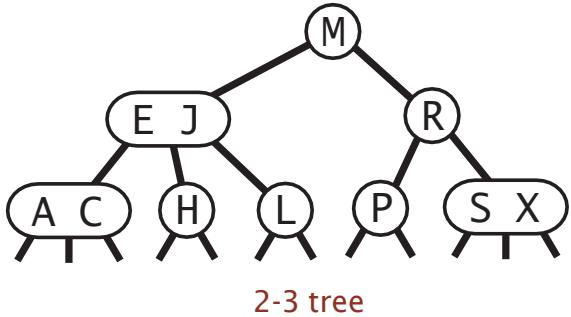
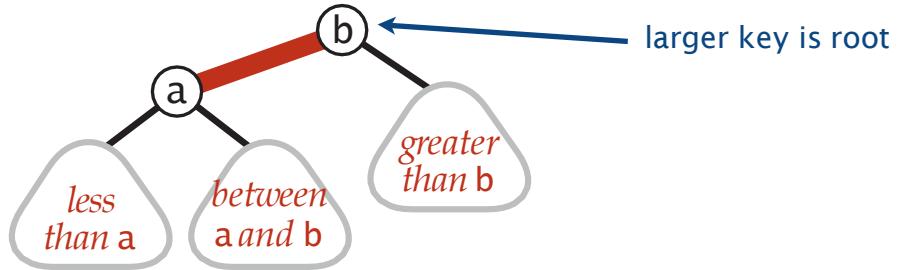
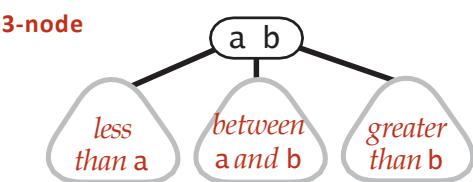
Tree height.

- Worst case: $\lg N$. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

Bottom line. Guaranteed **logarithmic** performance for search and insert.

Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

1. Represent 2–3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3-nodes.



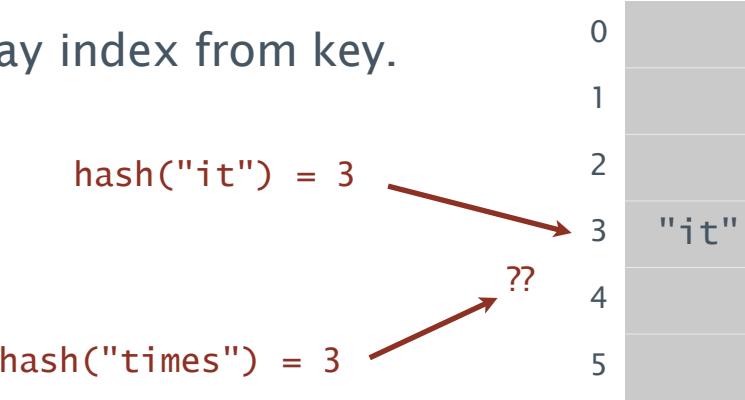
Hash tables

Save items in a **key-indexed table** (index is a function of the key).

Hash function. Method for computing array index from key.

Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.



Classic space-time tradeoff.

- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).

Symbol Table summary so far

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✓	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N^*$	$1.0 \lg N^*$	$1.0 \lg N^*$	✓	<code>compareTo()</code>

Symbol Table summary so far

Order of growth of the frequency of operations.

implementation	typical case			ordered operations	operations on keys
	search	insert	delete		
red-black BST	$\log N$	$\log N$	$\log N$	✓	<code>compareTo()</code>
hash table	1^\dagger	1^\dagger	1^\dagger		<code>equals()</code> <code>hashCode()</code>

† under uniform hashing assumption

use array accesses to make R-way decisions
(instead of binary decisions)

Q. Can we do better?



A. Yes, if we can avoid examining the entire key, as with string sorting.

Tries

- › Data structure for searching with string keys
- › From word “retrieval”, but read as “try” to be different than “tree”

Symbol Tables

- › Generic ST
- › <https://algs4.cs.princeton.edu/35applications/ST.java>

```
public class ST<Key extends Comparable<Key>, Value>
implements Iterable<Key> {

    private TreeMap<Key, Value> st;

    public void put(Key key, Value val) {...}

    public void delete(Key key) {...}

    public Value get(Key key) {...}

}
```

Symbol Tables

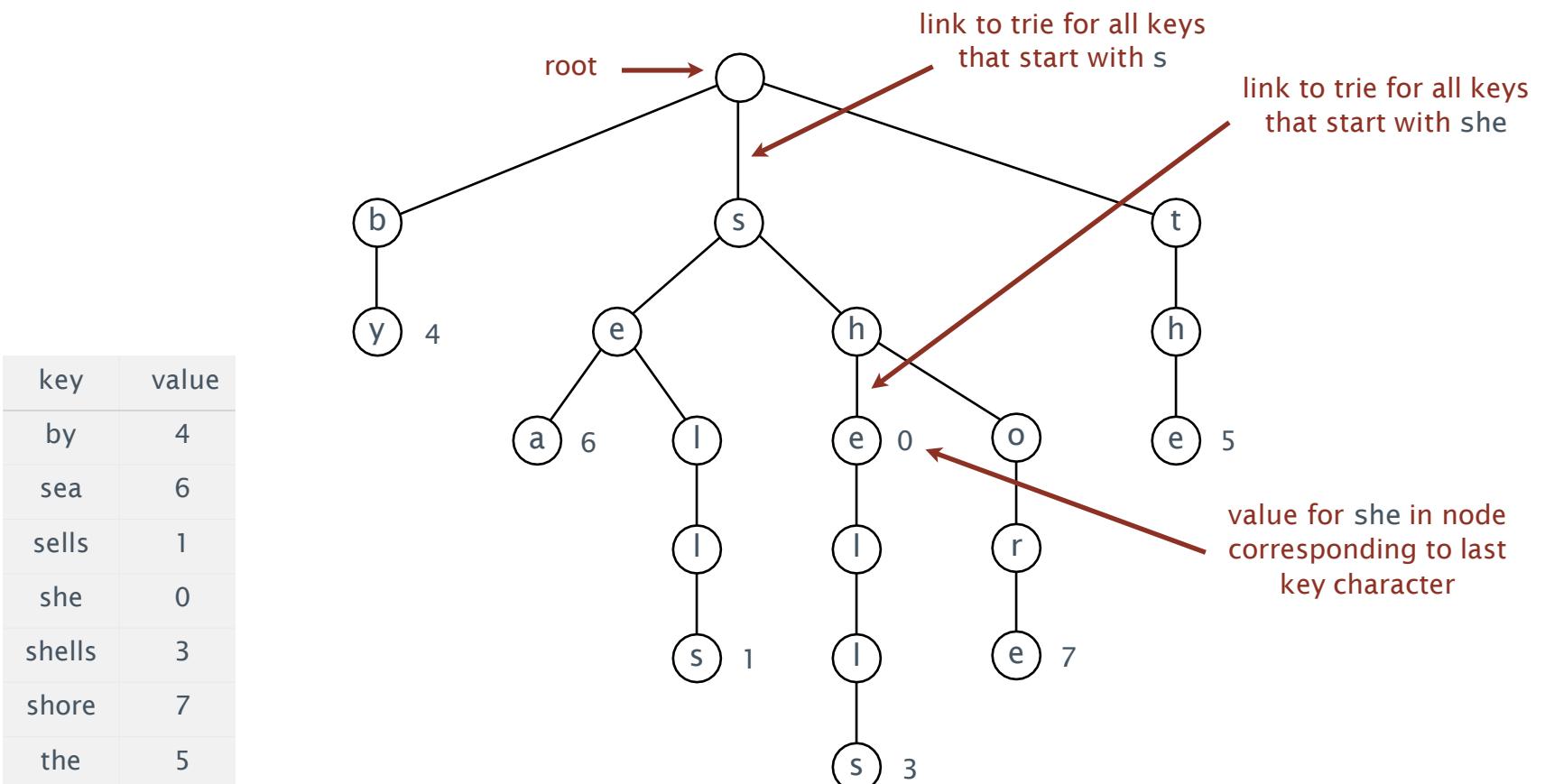
- › StringST

```
public class StringST<Value> {  
    public void put(String key, Value val) {...}  
    public void delete(String key) {...}  
    public Value get(String key) {...}  
}
```

Tries

Tries.

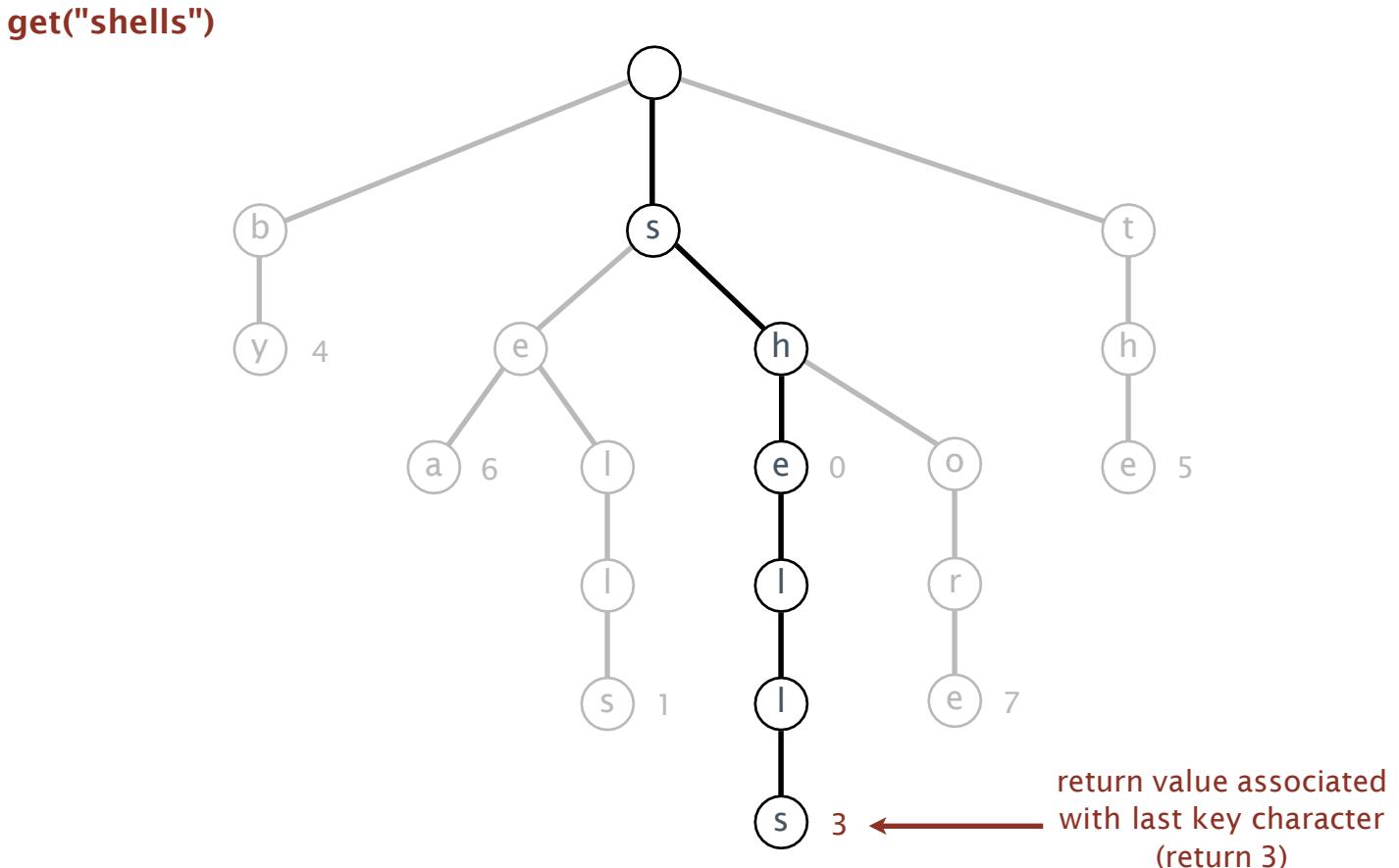
- Store characters in nodes (not keys).
- Each node has R children, one for each possible character.
(for now, we do not draw null links)



Search in a trie

Follow links corresponding to each character in the key.

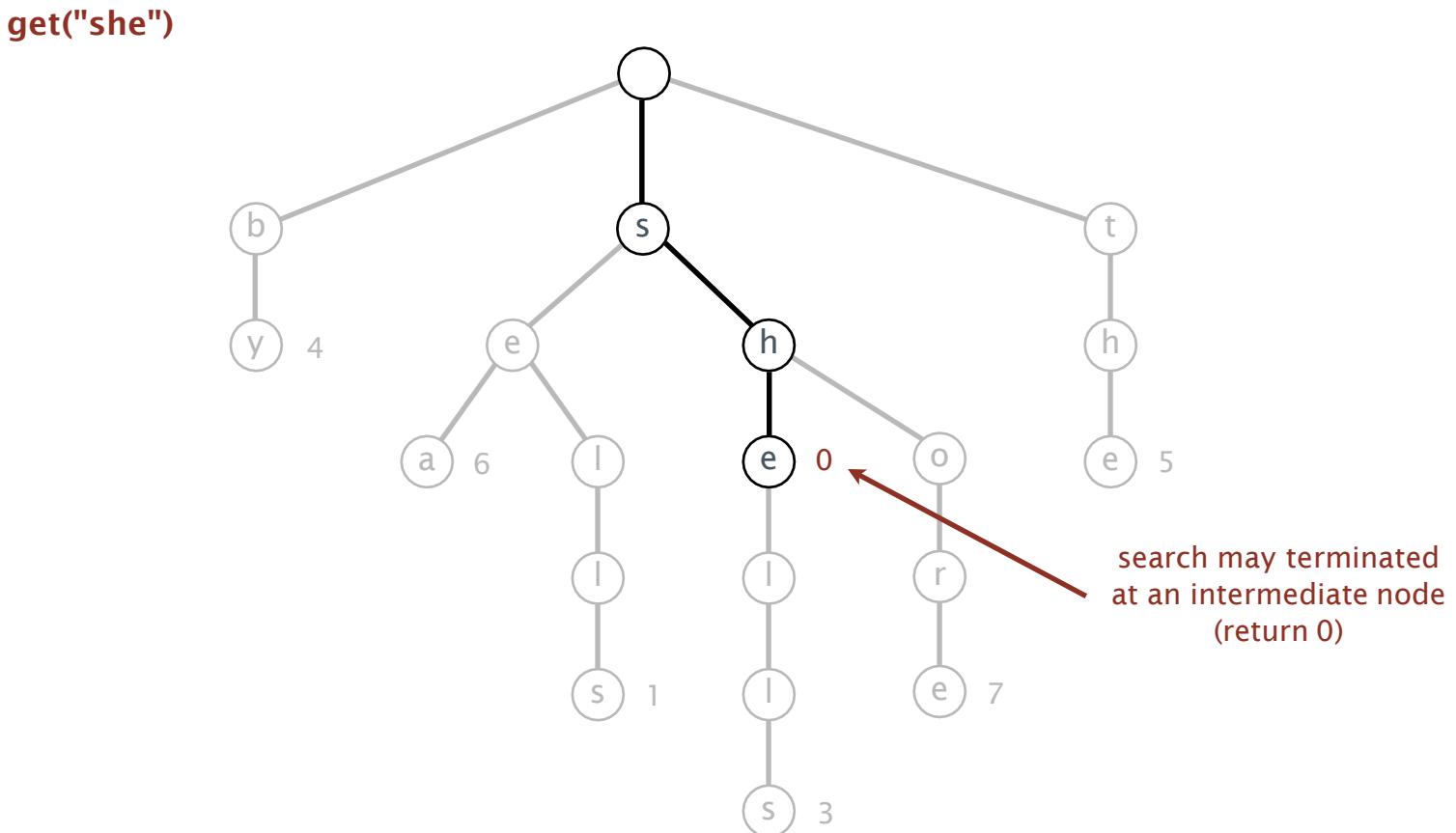
- **Search hit:** node where search ends has a non-null value.
- **Search miss:** reach null link or node where search ends has null value.



Search in a trie

Follow links corresponding to each character in the key.

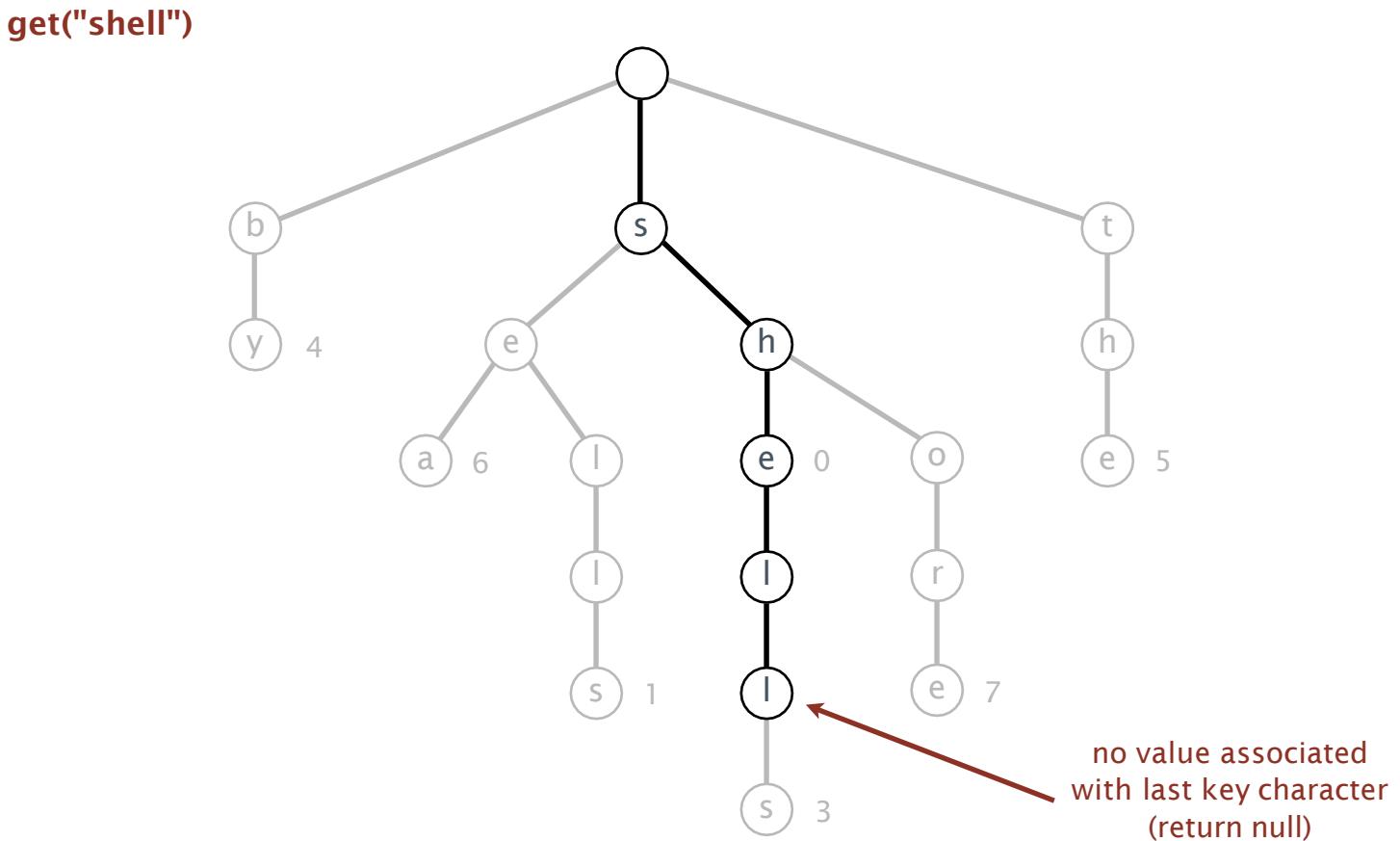
- **Search hit:** node where search ends has a non-null value.
- **Search miss:** reach null link or node where search ends has null value.



Search in a trie

Follow links corresponding to each character in the key.

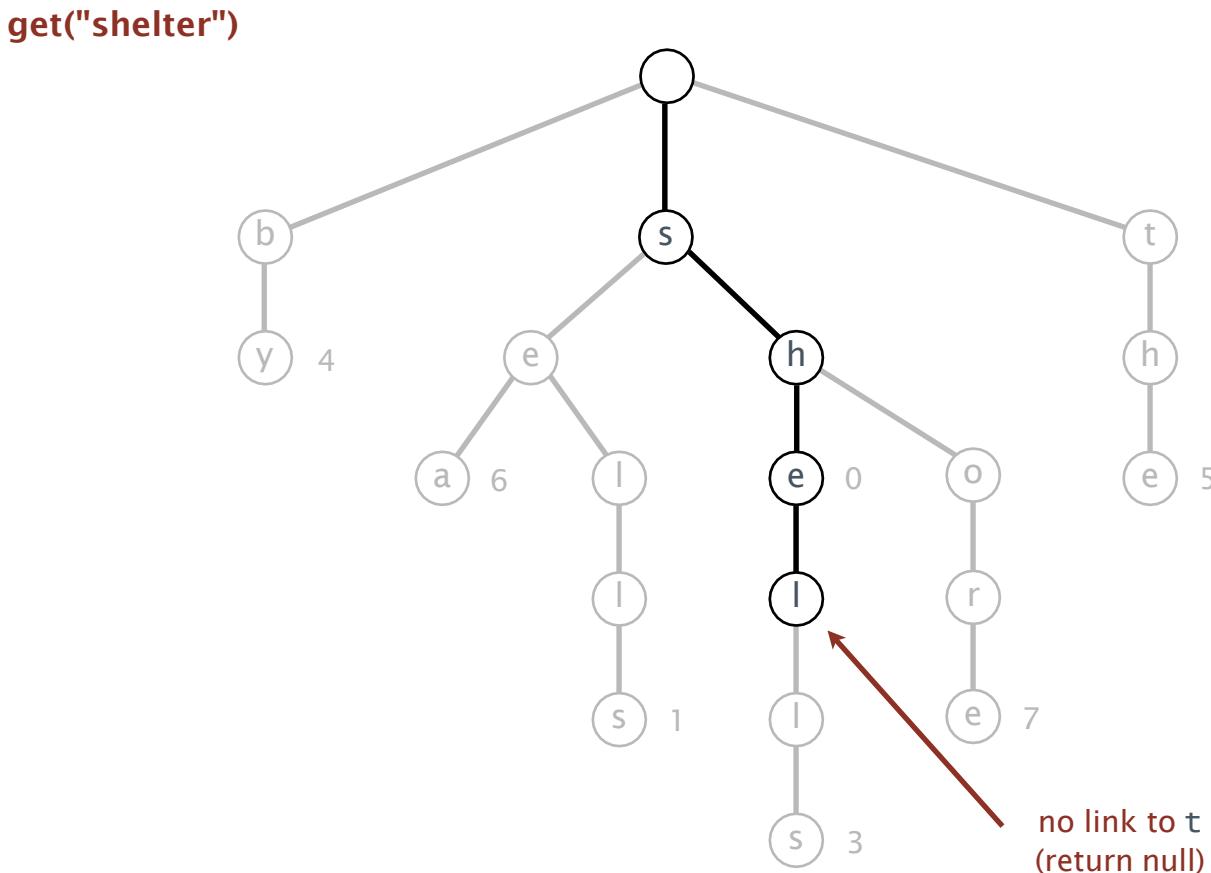
- Search hit: node where search ends has a non-null value.
- **Search miss:** reach null link or node where search ends has null value.



Search in a trie

Follow links corresponding to each character in the key.

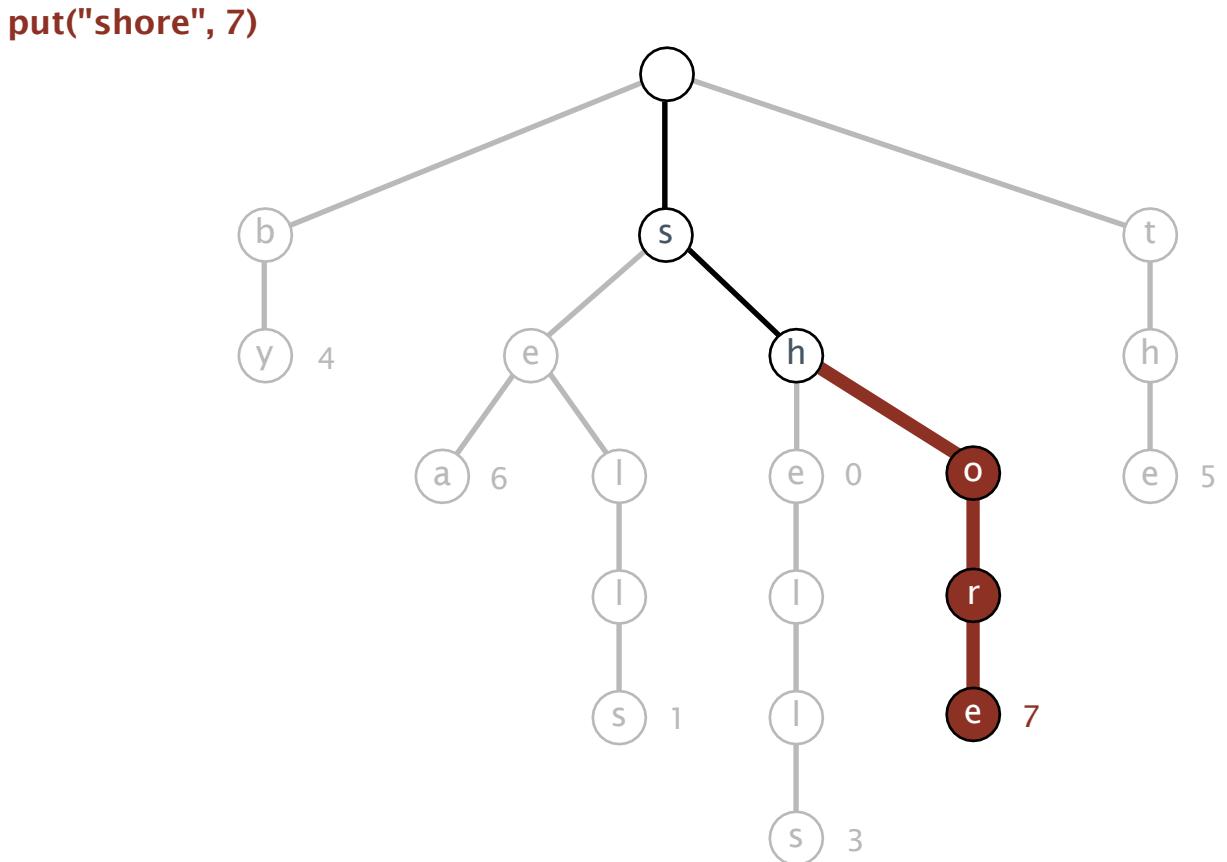
- Search hit: node where search ends has a non-null value.
- **Search miss:** reach null link or node where search ends has null value.



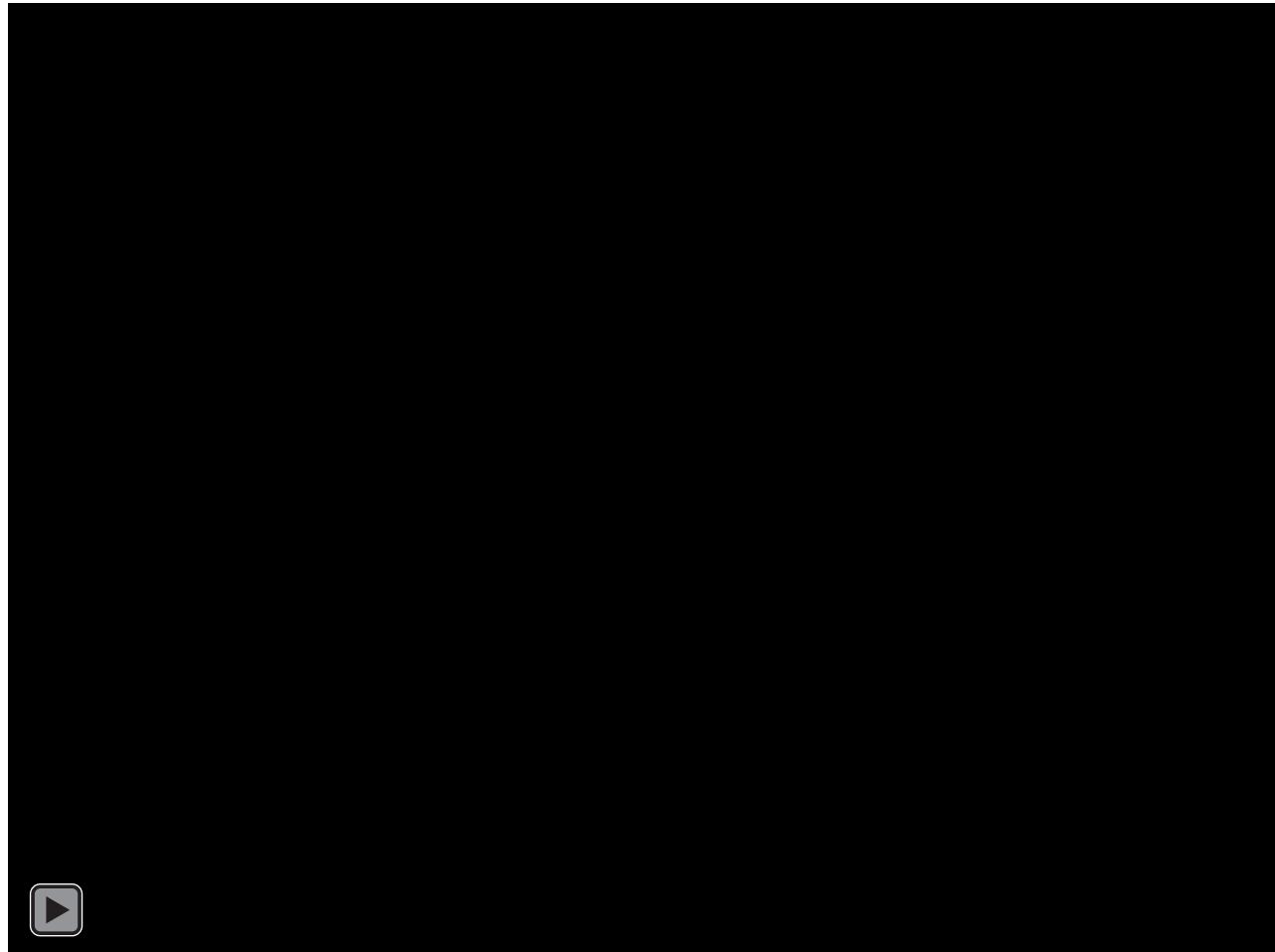
Insertion into a trie

Follow links corresponding to each character in the key.

- Encounter a null link: create new node.
- Encounter the last character of the key: set value in that node.

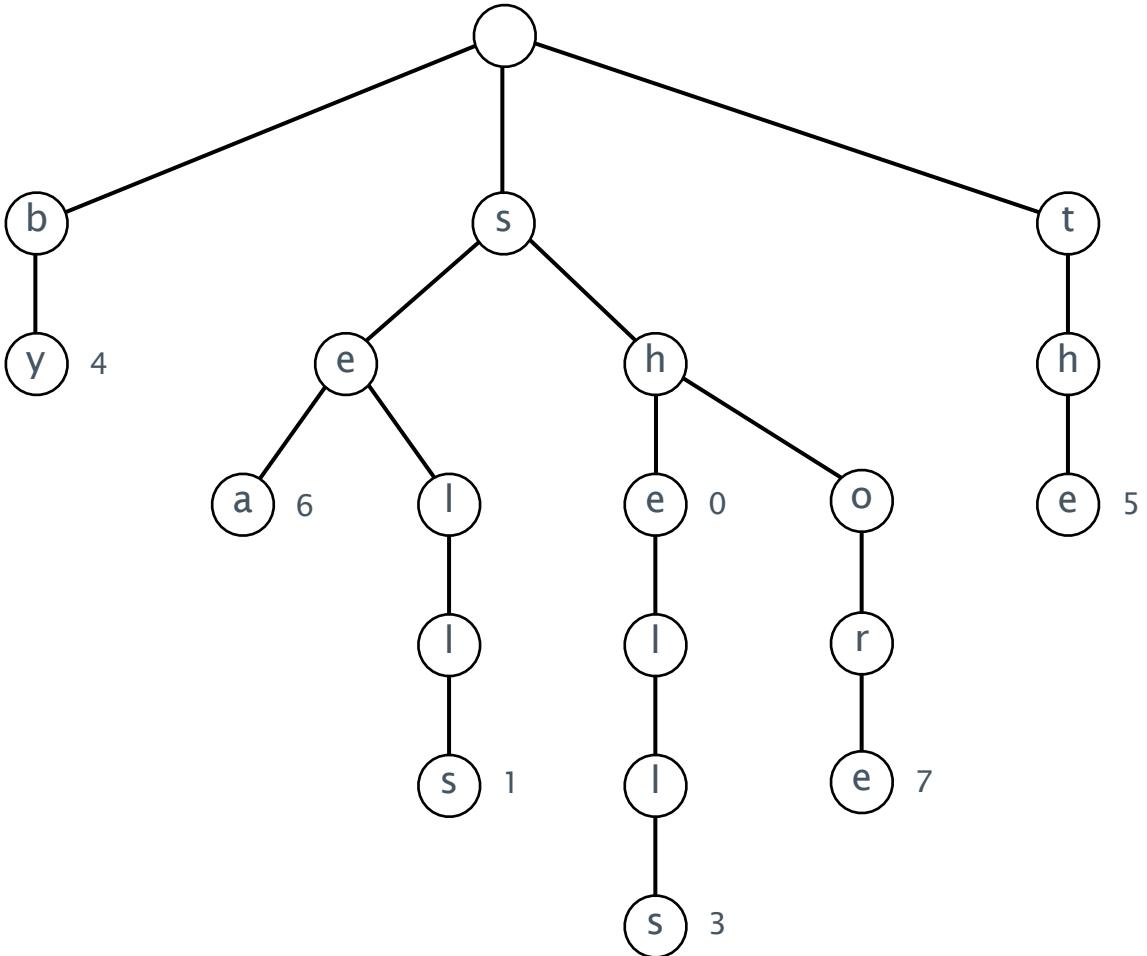


Trie construction demo



Trie construction demo

trie

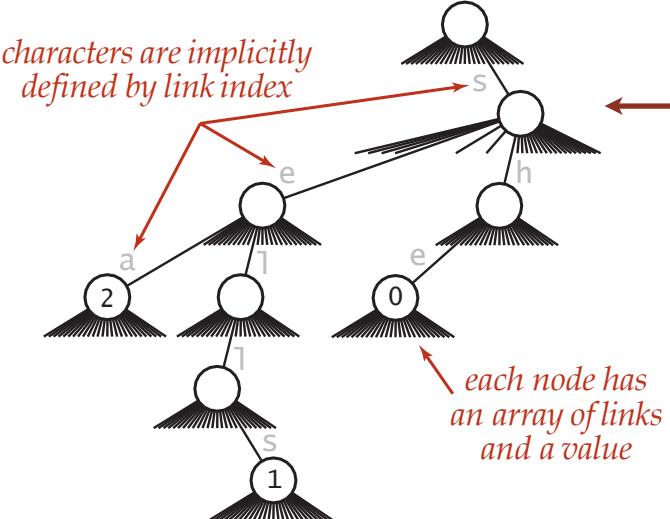
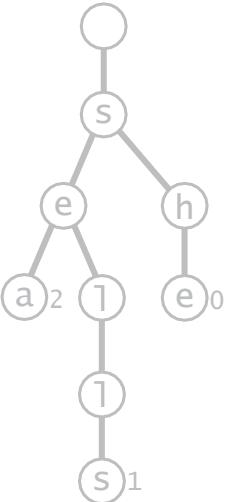


Trie representation: Java implementation

Node. A value, plus references to R nodes.

```
private static class Node
{
    private Object value;
    private Node[] next = new Node[R];
}
```

use Object instead of Value since no generic array creation in Java



R-way trie: Java implementation

```
public class TrieST<Value>
{
    private static final int R = 256;      ← extended ASCII
    private Node root = new Node();

    private static class Node
    { /* see previous slide */ }

    public void put(String key, Value val)
    { root = put(root, key, val, 0); }

    private Node put(Node x, String key, Value val, int d)
    {
        if (x == null) x = new Node();
        if (d == key.length()) { x.val = val; return x; }
        char c = key.charAt(d);
        x.next[c] = put(x.next[c], key, val, d+1);
        return x;
    }

    ...
}
```

R-way trie: Java implementation (continued)

```
:  
public boolean contains(String key)  
{  return get(key) != null;  }  
  
public Value get(String key)  
{  
    Node x = get(root, key, 0);  
    if (x == null) return null;  
    return (Value) x.val;  ← cast needed  
}  
  
private Node get(Node x, String key, int d)  
{  
    if (x == null) return null;  
    if (d == key.length()) return x;  
    char c = key.charAt(d);  
    return get(x.next[c], key, d+1);  
}  
}
```

Trie performance

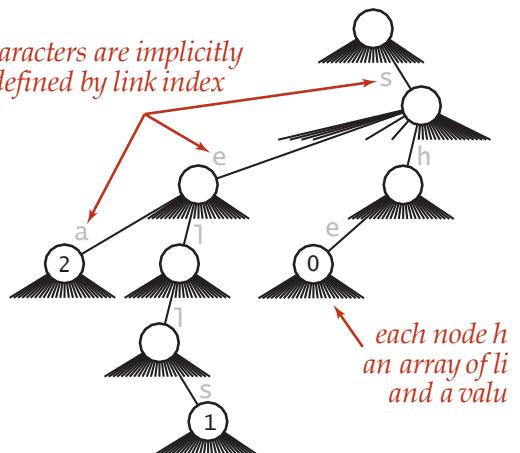
Search hit. Need to examine all L characters for equality.

Search miss.

- Could have mismatch on first character.
- Typical case: examine only a few characters (sublinear).

Space. R null links at each leaf.

(but sublinear space possible if many short strings share common prefixes)

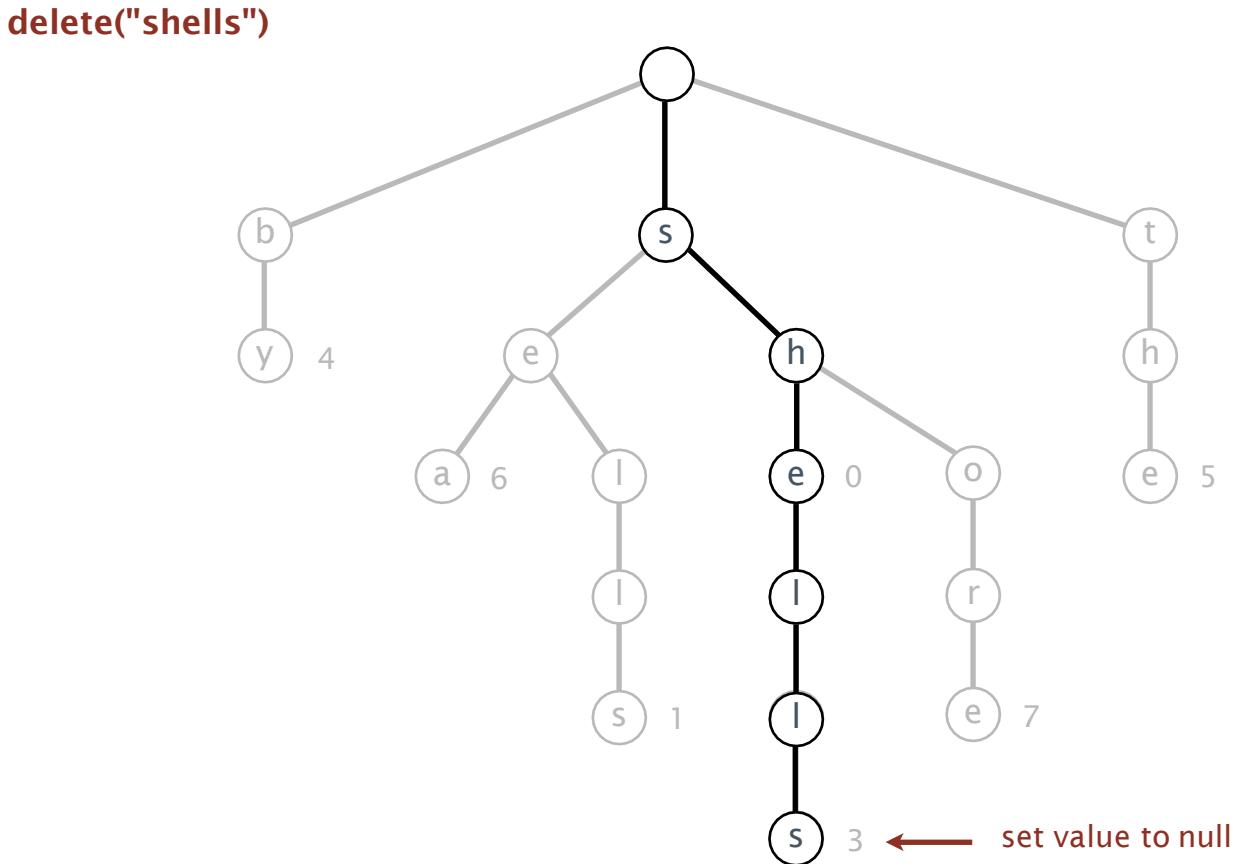


Bottom line. Fast search hit and even faster search miss, but wastes space.

Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

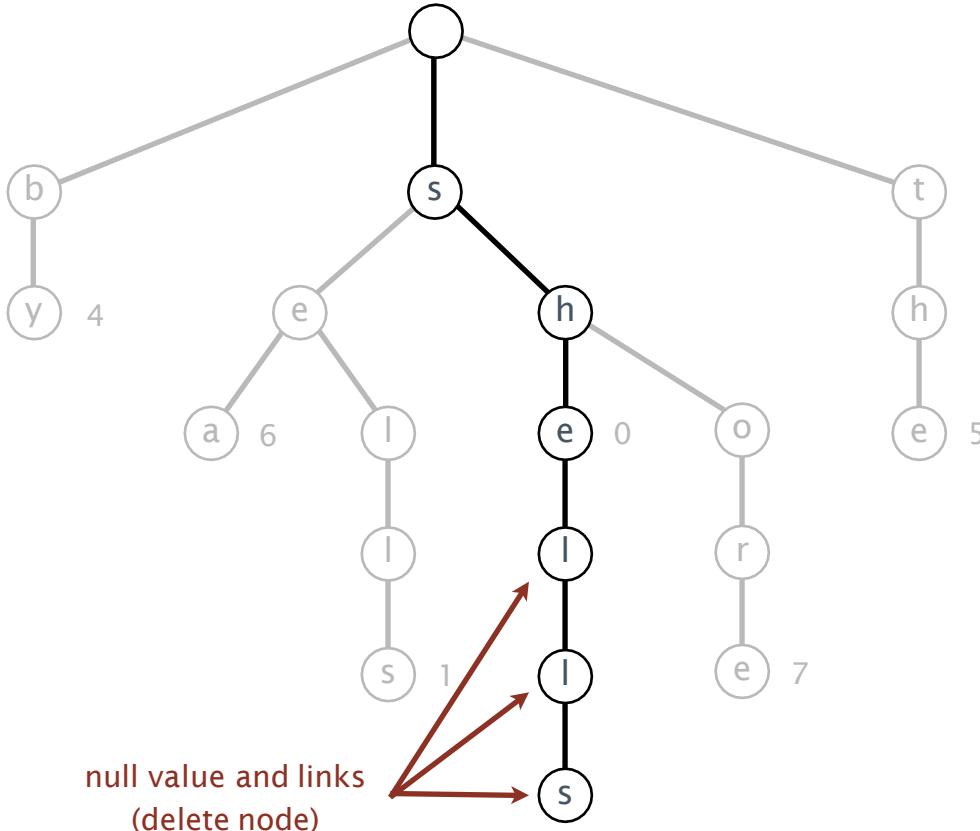


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")



Trie exercise

- › Construct a trie with following key-value pairs

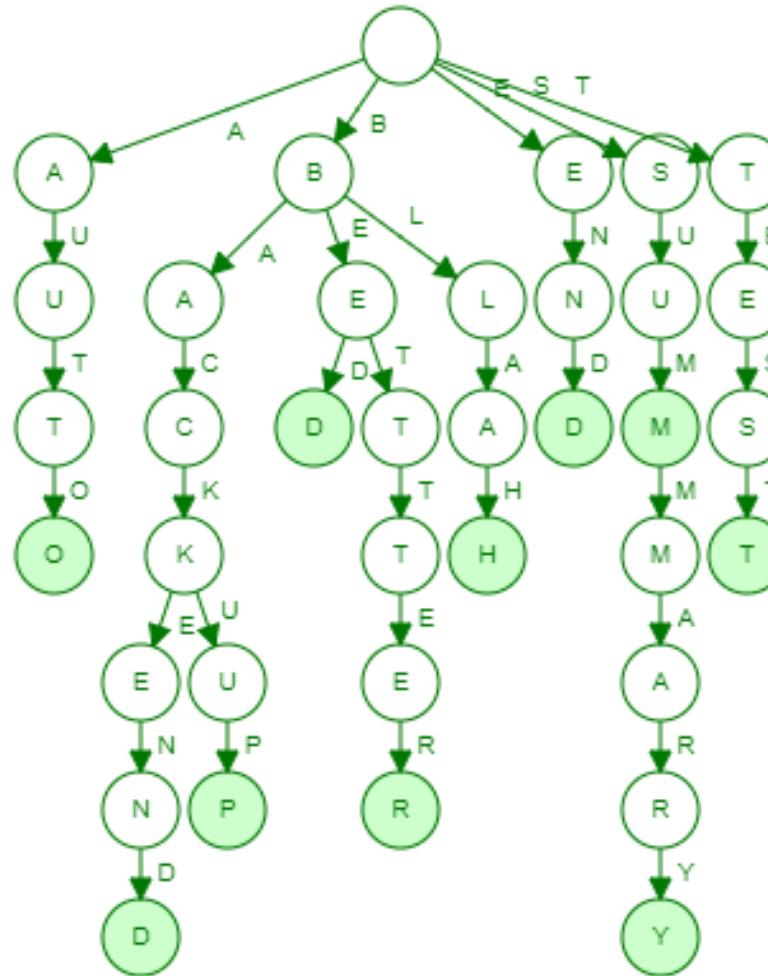
Key	value
bed	50
better	3
backend	30
backup	1
auto	1
test	3
summary	5
sum	40
end	3
blah	3

Trie exercise

Key	value
bed	50
better	3
backend	30
backup	1
auto	1
test	3
summary	5
sum	40
end	3
blah	3

- › Construct a trie with following key-value pairs
- › Assuming 26-digit radix (lower case letters), think about how many null links does the trie have?
- › Is there a more optimal way to do this?

<https://www.cs.usfca.edu/~galles/visualization/Trie.html>



String symbol table implementations cost summary

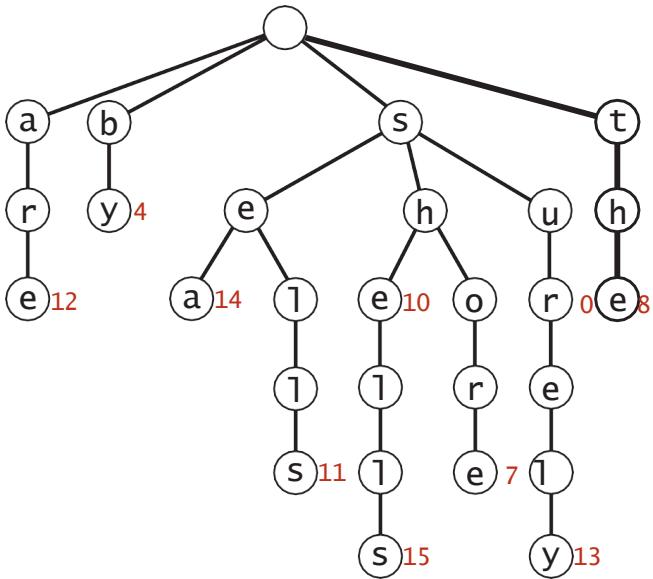
implementation	character accesses (typical case)				dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4
hashing (linear probing)	L	L	L	$4N \text{ to } 16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1)N$	1.12	<i>out of memory</i>

- Too much memory for large R .

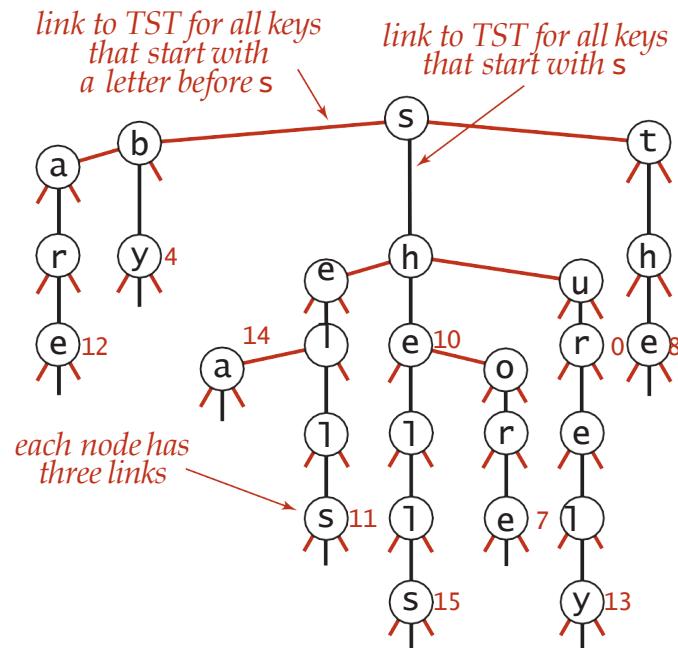
Ternary Search Tries (TSTs)

Ternary search tries

- Store characters and values in nodes (not keys).
- Each node has **3** children: smaller (left), equal (middle), larger (right).

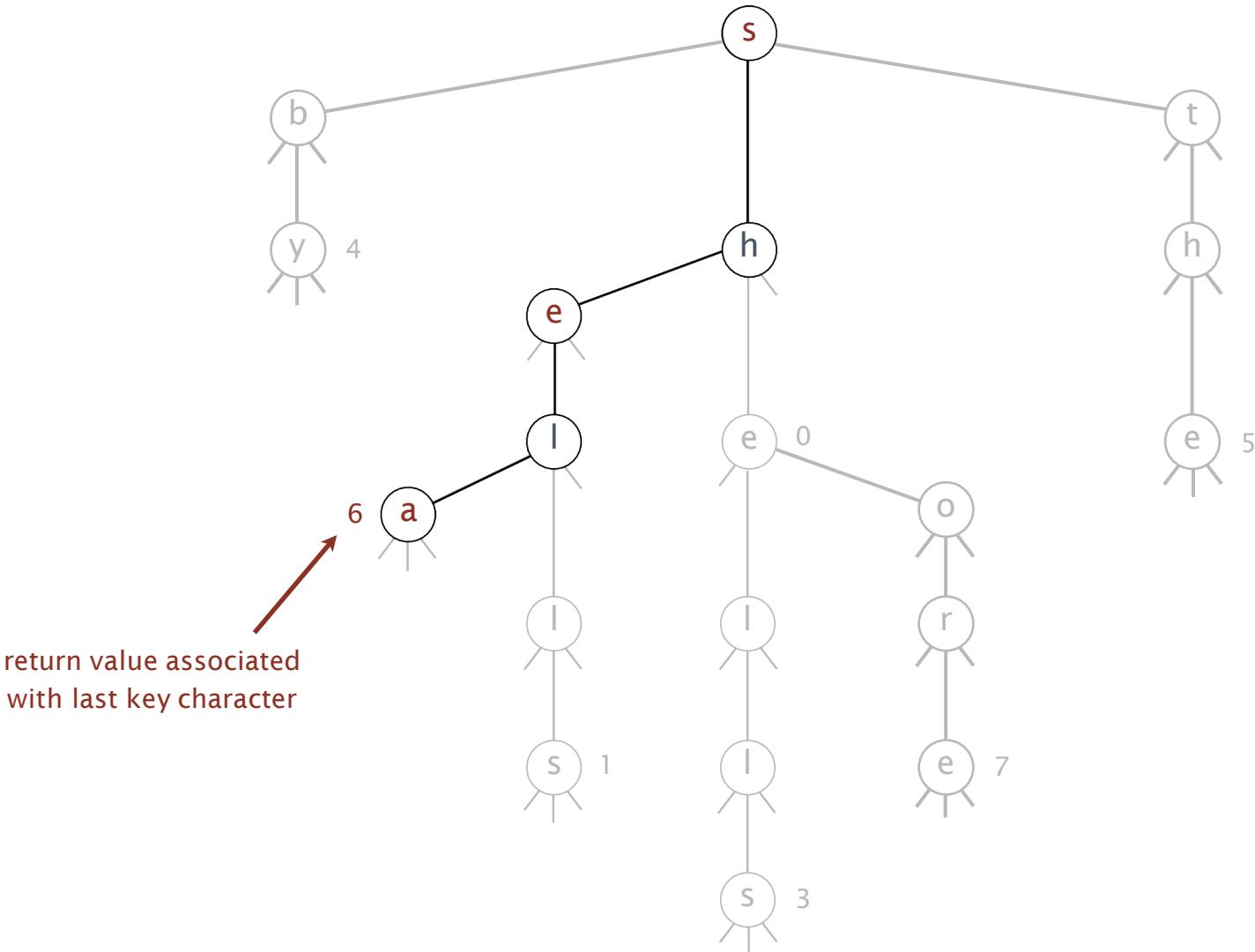


TST representation of a trie



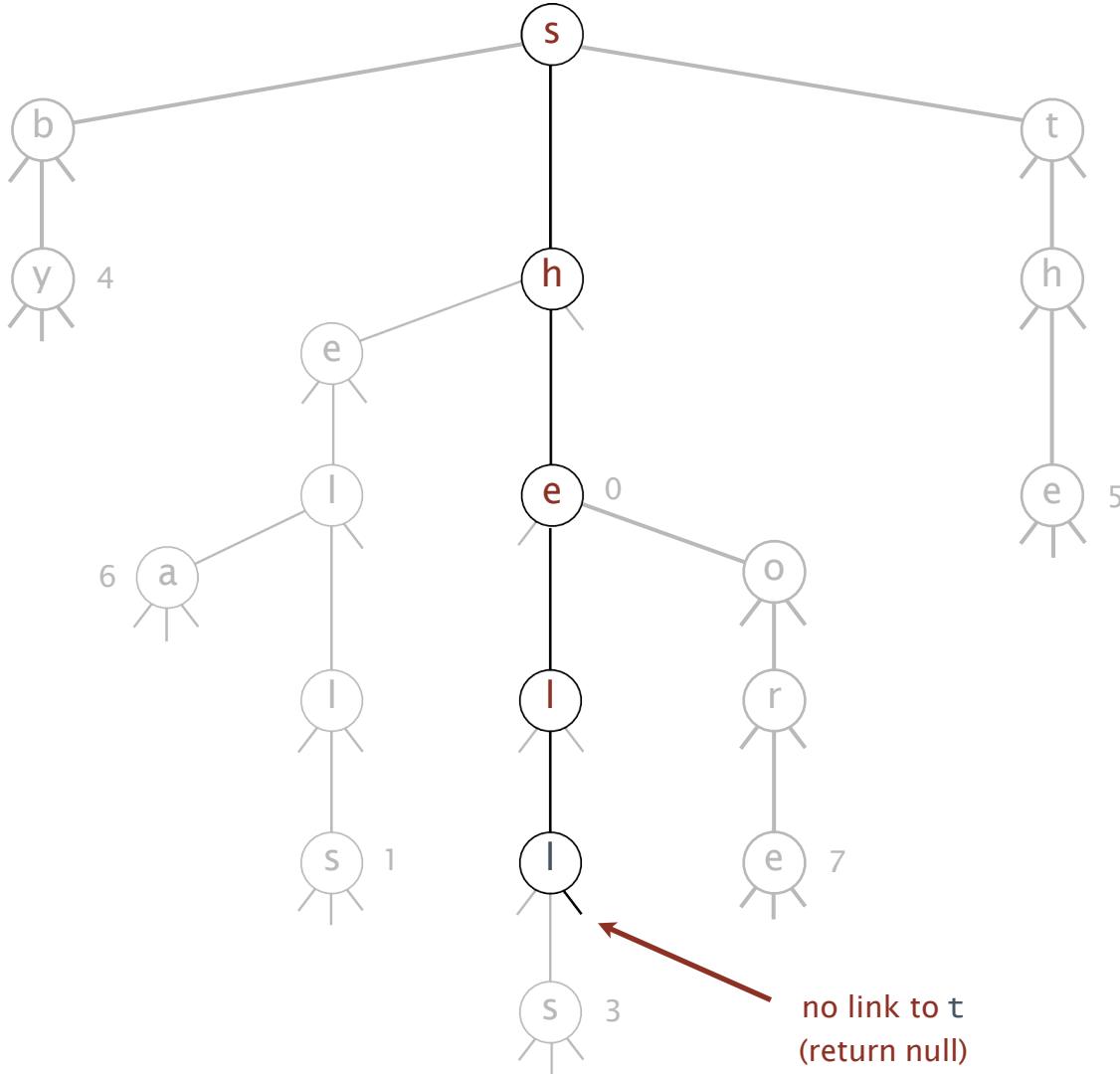
Search hit in a TST

get("sea")

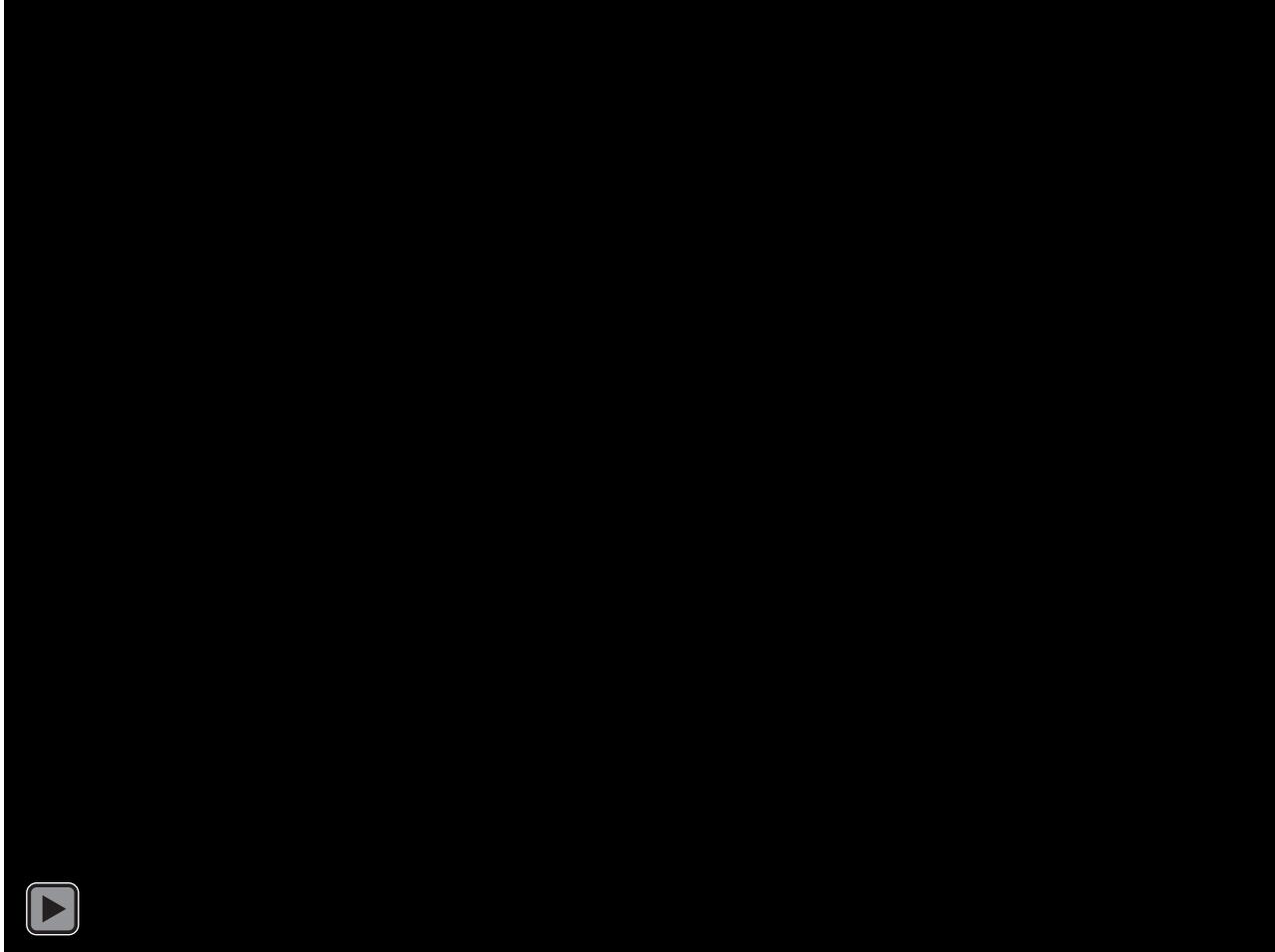


Search miss in a TST

get("shelter")

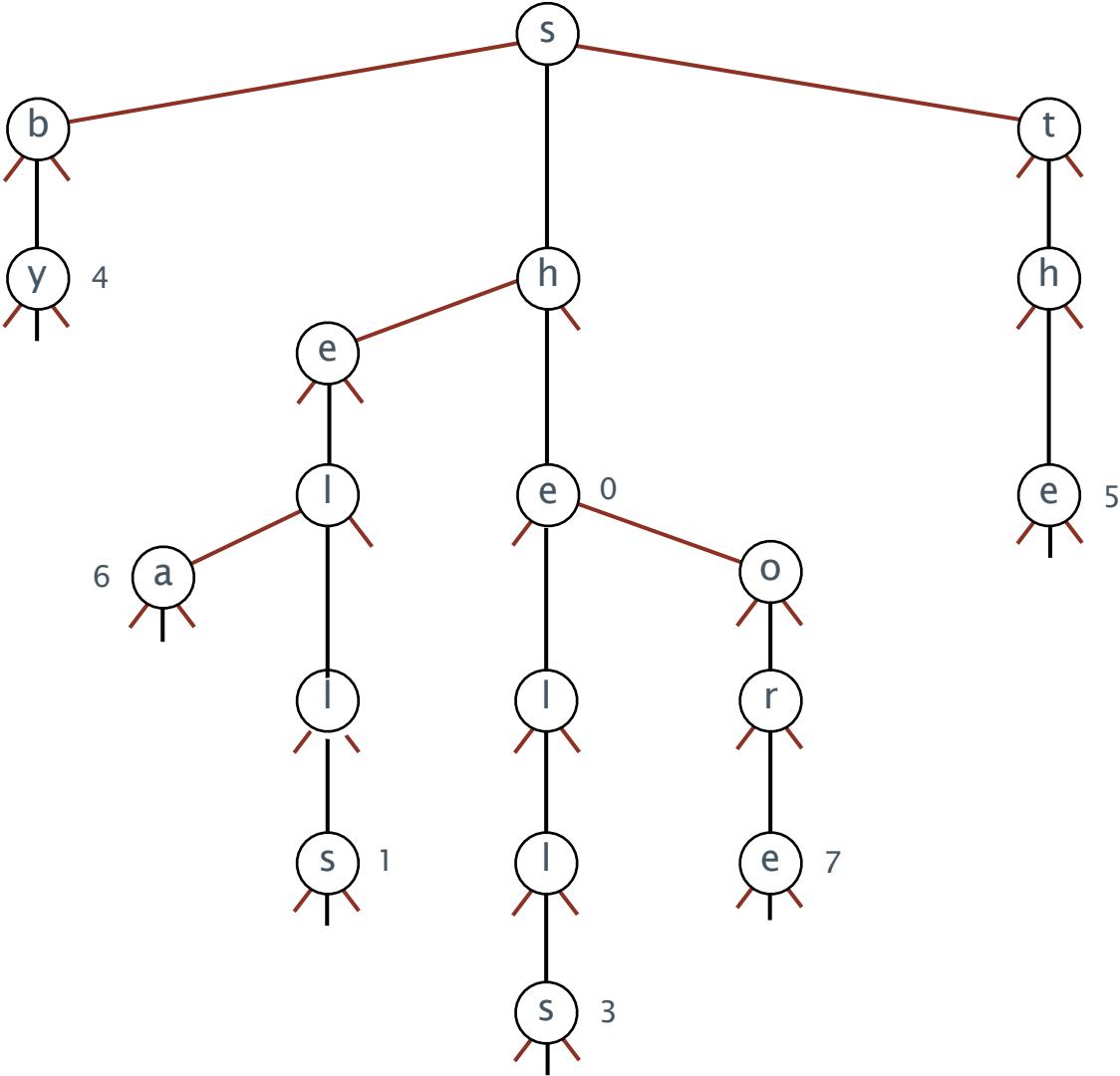


TST construction demo



Ternary search trie construction demo

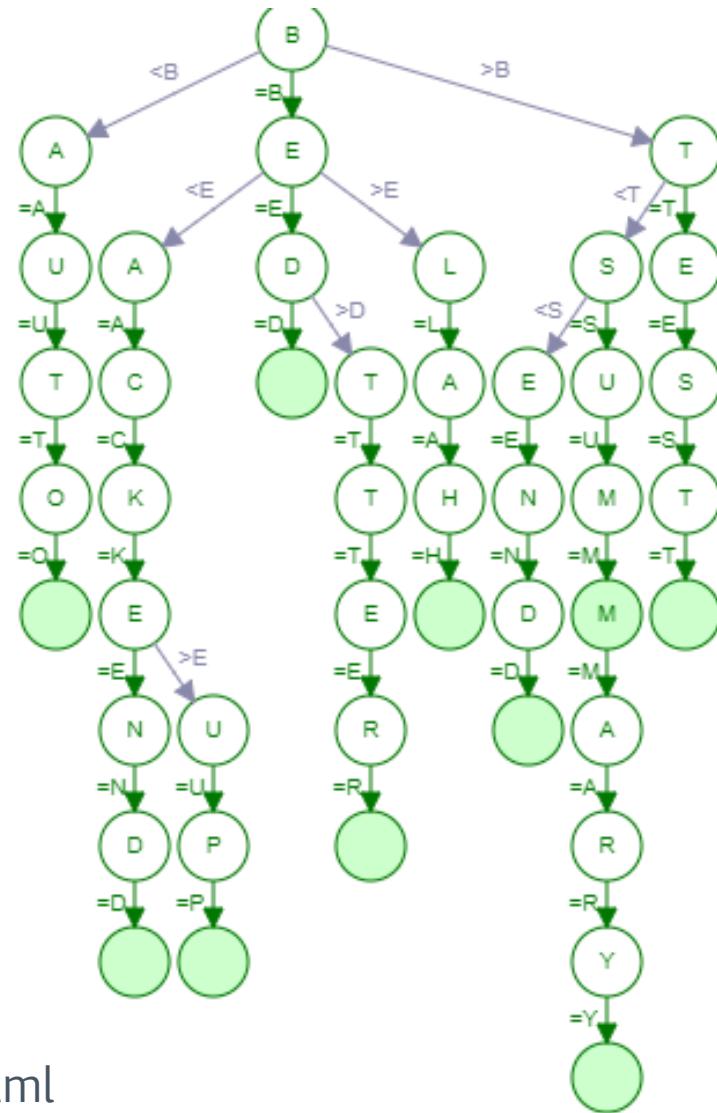
ternary search trie



› This time, build a TST using these keys and values

Key	value
bed	50
better	3
backend	30
backup	1
auto	1
test	3
summary	5
sum	40
end	3
blah	3

Key	value
bed	50
better	3
backend	30
backup	1
auto	1
test	3
summary	5
sum	40
end	3
blah	3



<https://www.cs.usfca.edu/~galles/visualization/TST.html>

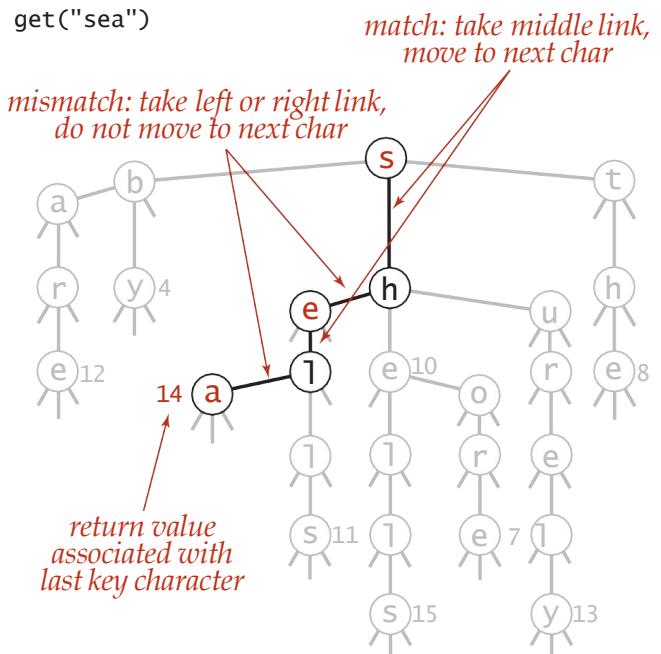
Search in a TST

Follow links corresponding to each character in the key.

- If less, take left link; if greater, take right link.
 - If equal, take the middle link and move to the next key character.

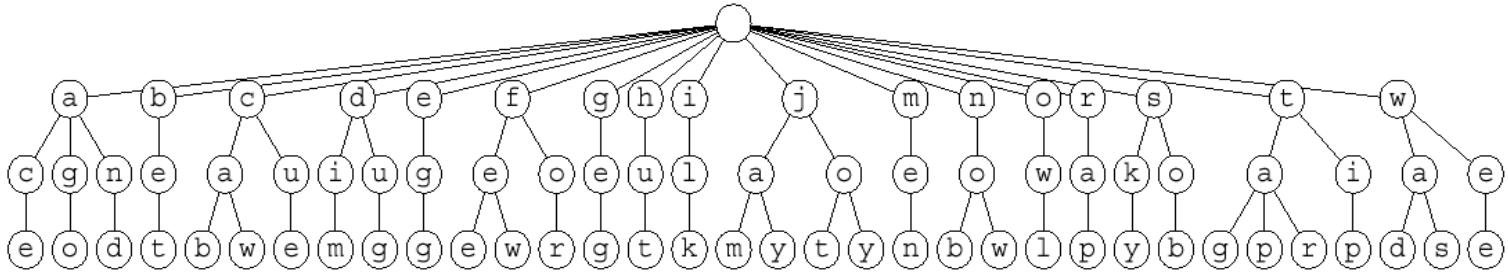
Search hit. Node where search ends has a non-null value.

Search miss. Reach a null link or node where search ends has null value.



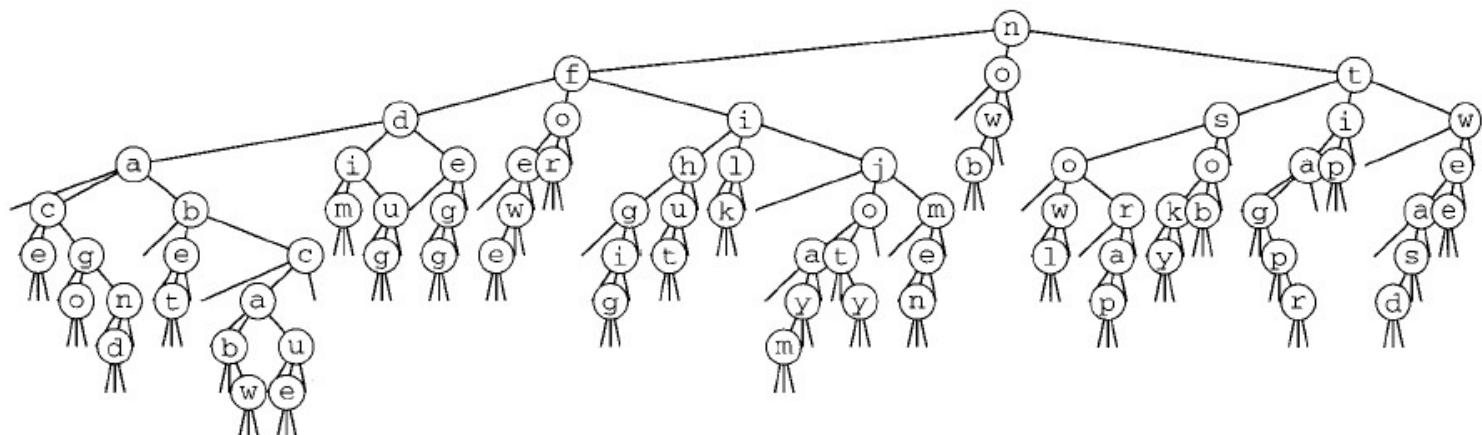
26-way trie vs. TST

26-way trie. 26 null links in each leaf.



26-way trie (1035 null links, not shown)

TST. 3 null links in each leaf.



TST (155 null links)

now
for
tip
ilk
dim
tag
jot
sob
nob
sky
hut
ace
bet
men
egg
few
jay
owl
joy
rap
gig
wee
was
cab
wad
caw
cue
fee
tap
ago
tar
jam
dug
and

String symbol table implementation cost summary

implementation	character accesses (typical case)				dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4
hashing (linear probing)	L	L	L	$4N \text{ to } 16N$	0.76	40.6
R-way trie	L	$\log_R N$	L	$(R+1)N$	1.12	<i>out of memory</i>
TST	$L + \ln N$	$\ln N$	$L + \ln N$	4 N	0.72	38.7

TST vs. hashing

Hashing.

- Need to examine entire key.
- Search hits and misses cost about the same.
- Performance relies on hash function.
- Does not support ordered symbol table operations.

TSTs.

- Works only for string (or digital) keys.
- Only examines just enough key characters.
- Search miss may involve only a few characters.
- Supports ordered symbol table operations (plus extras!).

Bottom line. TSTs are:

- Faster than hashing (especially for search misses).
- More flexible than red-black BSTs
 - supports character-based operations

String symbol table API

Character-based operations. The string symbol table API supports several useful character-based operations.

key	value
by	4
sea	6
sells	1
she	0
shells	3
shore	7
the	5

Prefix match. Keys with prefix sh: she, shells, and shore.

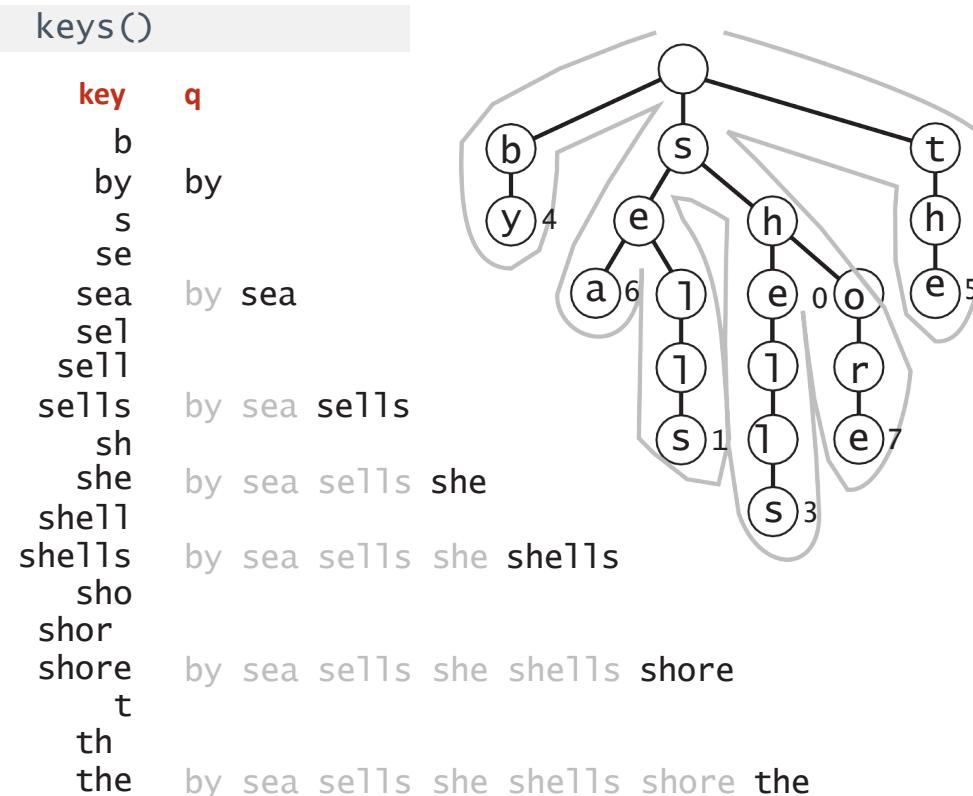
Wildcard match. Keys that match .he: she and the.

Longest prefix. Key that is the longest prefix of shellsort: shells.

Ordered iteration

To iterate through all keys in sorted order:

- Do inorder traversal of trie; add keys encountered to a queue.
- Maintain sequence of characters on path from root to node.



Applications -prefix matches

› Eg dropdown lists

The image shows two side-by-side Google search interfaces. Both feature a decorative banner at the top with the word 'GOOGLE' where the 'O' is replaced by a red heart containing a play button, and two white swans floating around it.

Left Search: The search bar contains the prefix 'why|'. Below the bar, a dropdown list shows suggestions starting with 'why':

- why him
- why
- why do dogs lick humans
- why is the sky blue
- why pancake tuesday
- why am i so tired
- whytes
- why did snapchat update
- why don't we
- why him cast

At the bottom of the search interface are three buttons: 'Google Search', 'I'm Feeling Lucky', and 'Learn more'.

Right Search: The search bar contains the prefix 'should i|'. Below the bar, a dropdown list shows suggestions starting with 'should i':

- should i buy bitcoin
- should i stay or should i go lyrics
- should i invest in bitcoin
- should i text him
- should i break up with him
- should i buy ripple
- should i invest in ripple
- should i buy litecoin
- should i buy ethereum
- should i get the flu vaccine

At the bottom of the search interface are three buttons: 'Google Search', 'I'm Feeling Lucky', and 'Learn more'. Below the right search interface is a small note: 'Report inappropriate predictions'.

Applications – prefix match

Ex. To send packet toward destination IP address, router chooses IP address in routing table that is longest prefix match.

"128"
"128.112"
"128.112.055"
"128.112.055.15"
"128.112.136"
"128.112.155.11"
"128.112.155.13"
"128.222"
"128.222.136"

represented as 32-bit
binary number for IPv4
(instead of string)

`longestPrefixOf("128.112.136.11") = "128.112.136"`
`longestPrefixOf("128.112.100.16") = "128.112"`
`longestPrefixOf("128.166.123.45") = "128"`

Symbol tables summary

A success story in algorithm design and analysis.

Red-black BST.

- Performance guarantee: $\log N$ key compares.
- Supports ordered symbol table API.

Hash tables.

- Performance guarantee: constant number of probes.
- Requires good hash function for key type.

Tries. R-way, TST.

- Performance guarantee: $\log N$ **characters** accessed.
- Supports character-based operations.

Bottom line. You can get at anything by examining 50-100 bits (!!?)

CSU22012: Data Structures and Algorithms II

Strings

Ivana.Dusparic@scss.tcd.ie

Outline of String algorithms

- › String sorting algorithms
 - Exploit properties of strings to speed up sorting
 - Radix sort
- › Tries
 - Data structures for searching with string keys – more efficient than general purpose ones previously discussed, eg hashtables, search trees
- › Substring search
 - Interesting problem- multiple approaches illustrating different algorithm design techniques
- › Regular expressions
 - Searching for incomplete patterns rather than exact substrings
- › Data compression
 - Save storage, faster network transfer
 - Run length encoding, Huffman compression

What are strings?

- › Sequences of characters
 - Text
 - Genome sequences
- › What are characters then?
 - In C – char data type is 8 bit integer, 7-bit ascii, 256 characters max
 - In Java – char data type, 16 bit unsigned int, range ‘\u0000’ to ‘\uffff’ (0 to 65,535)

java.lang.String

- › Immutable sequence of characters
- › Implements Comparable
- › Implements CharSequences
 - Methods length() – number of characters
 - charAt(i) – returns the i-th character
 - Concatenation- concatenate one string to the end of another
java.lang.String
 - Substring – extract a subsequence of characters

Immutable?

- › String cannot be modified
- › What happens here then?

```
String test = "blah";
```

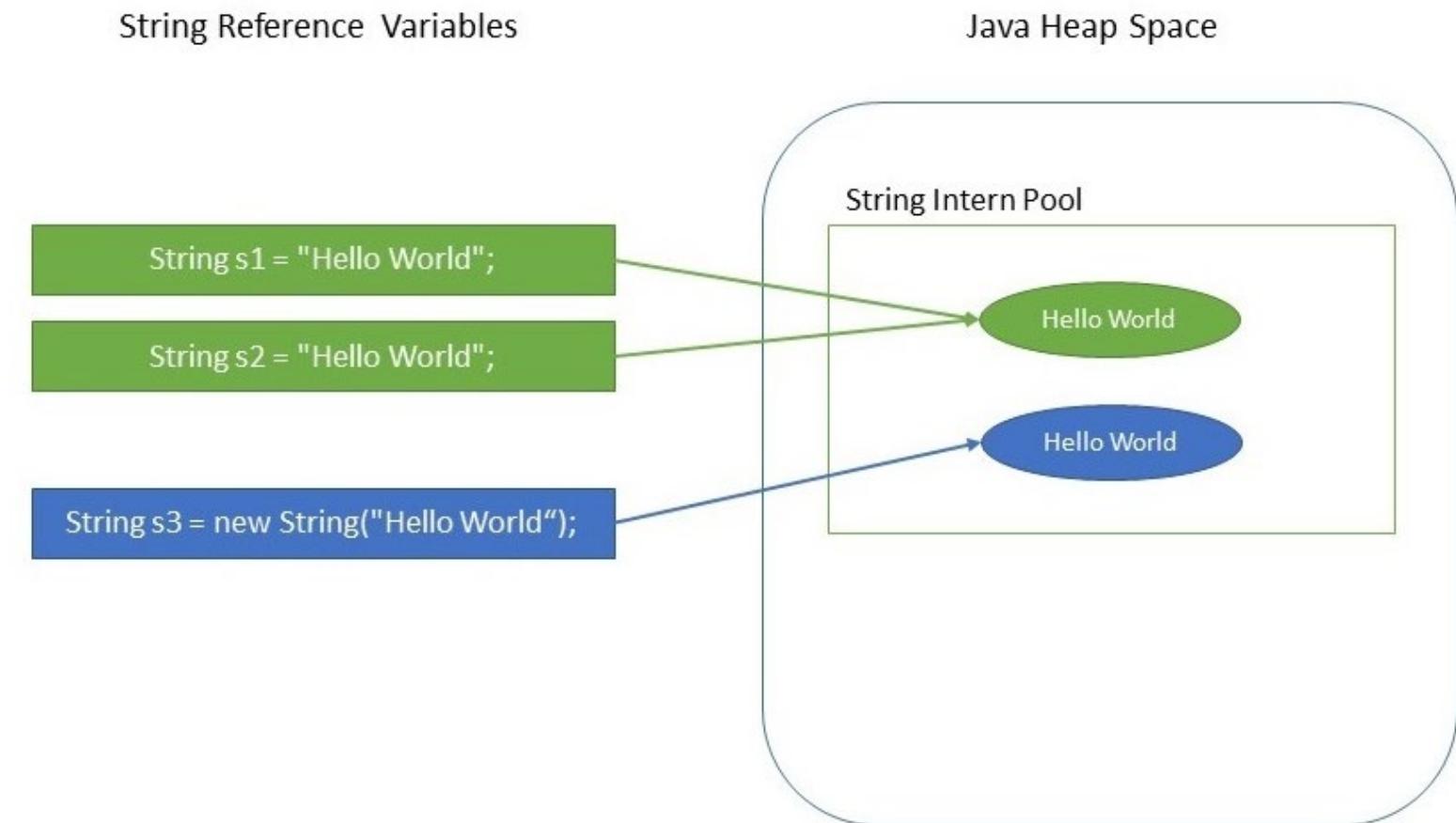
```
test = "meh";
```

```
System.out.println(test); //what does this print?
```

New string is created and reference, test now points to it instead of the old one – reference to “blah” text has been lost, it’s still unchanged somewhere in memory

Memory leaks/garbage collection

String pool



<https://www.baeldung.com/java-string-immutable>

Other implications of immutability

- › Security
 - Parameters in many methods which could introduce vulnerability
 - security threats, eg network connection is passed a string – it could be modified to connect to a different machine, or a modified file name can be passed in etc
- › Thread-safe
 - No need for synchronisation if shared between threads – no thread can modify it
- › Can be used as keys in symbol tables
- › Can calculate and save hashCode – efficiency
- › <https://www.baeldung.com/java-string-immutable>

String implementation

char [] value

int length - saved for efficiency so it doesn't have to be calculated every time we need it

int hash - calculated as $s[0]*31^{n-1} + s[1]*31^{n-2} + \dots + s[n-1]$

Cost of String operations

- › How long does it take to reverse a String?

```
public static String reverse(String s)
{
    String rev = "";
    for (int i = s.length() - 1; i >= 0; i--)
        rev += s.charAt(i);
    return rev;
}
```

← quadratic time

- › Alternative – mutable sequence of characters, StringBuilder

```
public static String reverse(String s)
{
    StringBuilder rev = new StringBuilder();
    for (int i = s.length() - 1; i >= 0; i--)
        rev.append(s.charAt(i));
    return rev.toString();
}
```

← linear time

Substring operation

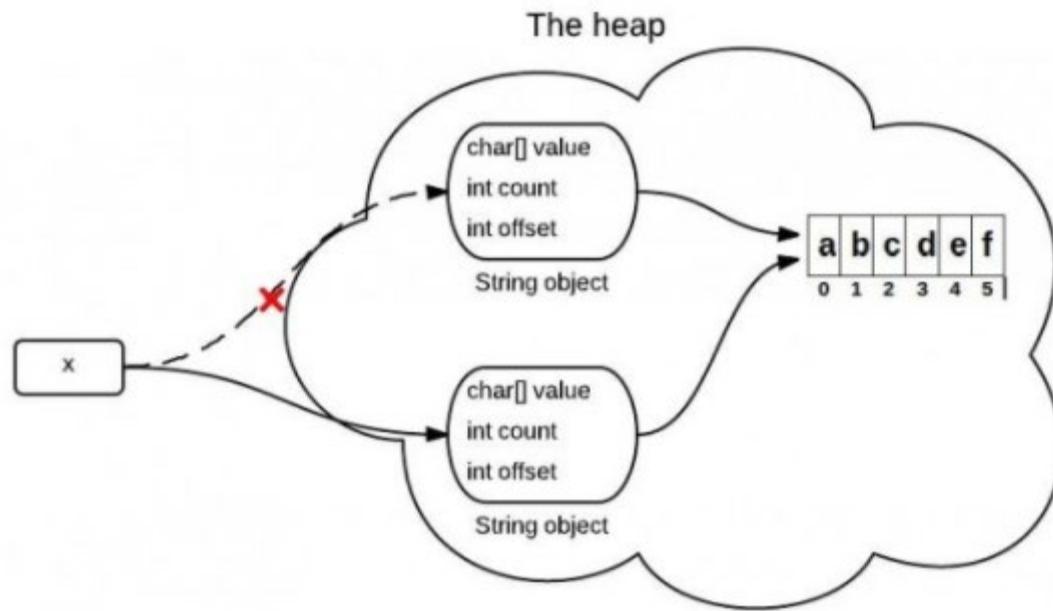
- › Java 6 vs Java 7 (and up)

```
String x = "abcdef";  
x = x.substring(1,3);  
System.out.println(x);
```

<https://www.programcreek.com/2013/09/the-substring-method-in-jdk-6-and-jdk-7/>

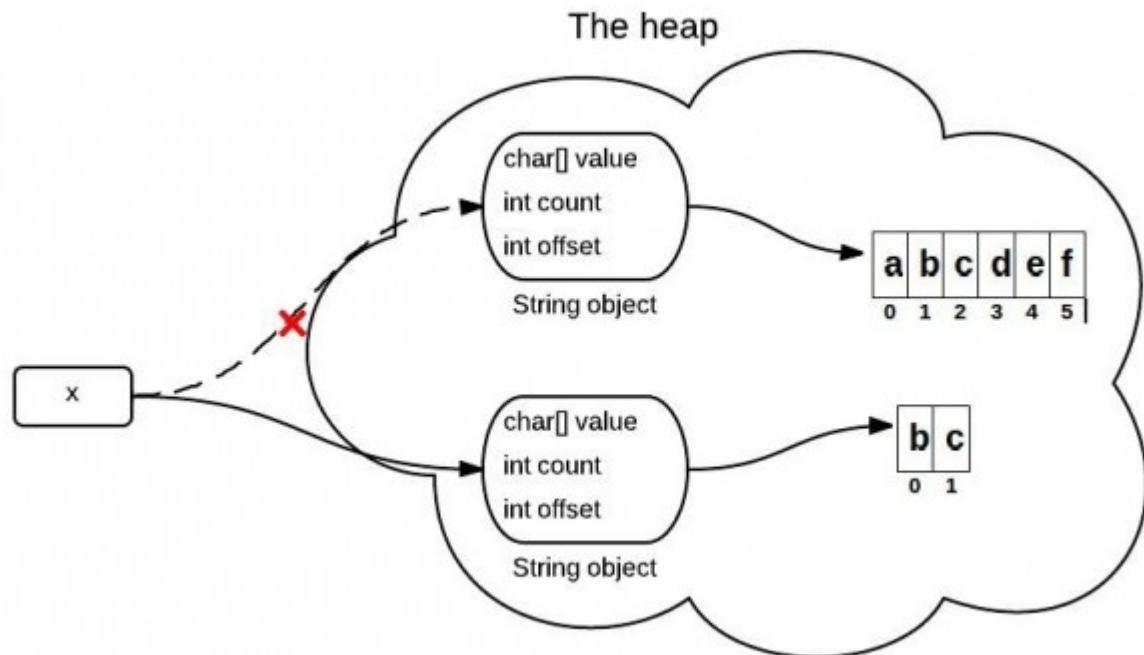
Substring operation in Java 6

- › Cheap (cost of 1)
- › Memory leaks – unused portion of the string can't be garbage collected
- › Points to same value
- › Count modified
- › Offset modified



Substring operation in Java 7 (now)

- › Cost of N, new string needs to be created- characters copied into it
- › Old string can be garbage collected



Comparing 2 Strings

Q. How many character compares to compare two strings of length W ?

p	r	e	f	e	t	c	h
0	1	2	3	4	5	6	7
p	r	e	f	i	x	e	s

Running time. Proportional to length of longest common prefix.

- Proportional to W in the worst case.
- But, often sublinear in W .

Different Alphabets

- › Performance depends on the size of the alphabet (i.e., unique characters)

Digital key. Sequence of digits over fixed alphabet.

Radix. Number of digits R in alphabet.

name	R_0	$\lg R_0$	characters
BINARY	2	1	01
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
DNA	4	2	ACTG
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
PROTEIN	20	5	ACDEFGHIJKLMNOPQRSTUVWXYZ
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
ASCII	128	7	ASCII characters
EXTENDED_ASCII	256	8	extended ASCII characters
UNICODE16	65536	16	Unicode characters

Key-Indexed Counting

Review: summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✓	compareTo()
quicksort	$1.39 N \lg N^*$	$1.39 N \lg N$	$c \lg N$		compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1		compareTo()

* probabilistic

Lower bound. $\sim N \lg N$ compares required by any compare-based algorithm.

Q. Can we do better (despite the lower bound)?

A. Yes, if we don't depend on key compares. ← use array accesses
to make R-way decisions
(Instead of binary decisions)

Key-indexed counting

- › Basis for other more complex sorting algorithms
 - LSD and MSD (least and most significant digit)
- › Specialized sorting algorithm which works best when the following conditions are met:
 - Input consists of collection of n items
 - Maximum possible value of each of the individual item is K

Key-indexed counting: assumptions about keys

Assumption. Keys are integers between 0 and $R - 1$.

Implication. Can use key as an array index.

Applications.

- Sort string by first letter.
- Sort class roster by section.
- Sort phone numbers by area code.
- Subroutine in a sorting algorithm. [stay tuned]

Remark. Keys may have associated data \Rightarrow
can't just count up number of keys of each value.

Input		sorted result (by section)
name	section	
Anderson	2	Harris 1
Brown	3	Martin 1
Davis	3	Moore 1
Garcia	4	Anderson 2
Harris	1	Martinez 2
Jackson	3	Miller 2
Johnson	4	Robinson 2
Jones	3	White 2
Martin	1	Brown 3
Martinez	2	Davis 3
Miller	2	Jackson 3
Moore	1	Jones 3
Robinson	2	Taylor 3
Smith	4	Williams 3
Taylor	3	Garcia 4
Thomas	4	Johnson 4
Thompson	4	Smith 4
White	2	Thomas 4
Williams	3	Thompson 4
Wilson	4	Wilson 4

↑
keys are
small integers

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

$R = 6$

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	
0	d	
1	a	use a for 0
2	c	b for 1
3	f	c for 2
4	f	d for 3
5	b	e for 4
6	d	f for 5
7	b	
8	f	
9	b	
10	e	
11	a	

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

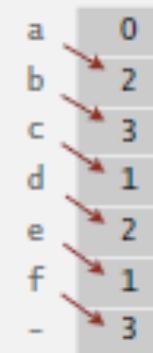
- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;
    
for (int r = 0; r < R; r++)
    count[r+1] += count[r];
    
for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];
    
for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

count
requencies

i	a[i]	offset by 1 [stay tuned]	r	count[r]
0	d		a	0
1	a		b	2
2	c		c	3
3	f		d	1
4	f		e	2
5	b		f	1
6	d		-	3
7	b			
8	f			
9	b			
10	e			
11	a			



Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r]; compute cumulates → count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	r	count[r]
0	d		
1	a		
2	c		
3	f	a	0
4	f	b	2
5	b	c	5
6	d	d	6
7	b	e	8
8	f	f	9
9	b		
10	e		
11	a		

6 keys < d, 8 keys < e
so d's go in a[6] and a[7]

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move Items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	a
2	c	r count[r]	b
3	f	a	b
4	f	b	b
5	b	c	c
6	d	d	d
7	b	e	d
8	f	f	e
9	b	-	f
10	e	12	f
11	a	12	f

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

copy back → for (int i = 0; i < N; i++)
               a[i] = aux[i];
```

i	a[i]	r	count[r]	i	aux[i]
0	a			0	a
1	a			1	a
2	b			2	b
3	b	a	2	3	b
4	b	b	5	4	b
5	c	c	6	5	c
6	d	d	8	6	d
7	d	e	9	7	d
8	e	f	12	8	e
9	f	-	12	9	f
10	f			10	f
11	f			11	f

Key-indexed counting: analysis

Proposition. Key-indexed takes time proportional to $N + R$.

Proposition. Key-indexed counting uses extra space proportional to $N + R$.

Stable? ✓

a[0]	Anderson	2	Harris	1	aux[0]
a[1]	Brown	3	Martin	1	aux[1]
a[2]	Davis	3	Moore	1	aux[2]
a[3]	Garcia	4	Anderson	2	aux[3]
a[4]	Harris	1	Martinez	2	aux[4]
a[5]	Jackson	3	Miller	2	aux[5]
a[6]	Johnson	4	Robinson	2	aux[6]
a[7]	Jones	3	White	2	aux[7]
a[8]	Martin	1	Brown	3	aux[8]
a[9]	Martinez	2	Davis	3	aux[9]
a[10]	Miller	2	Jackson	3	aux[10]
a[11]	Moore	1	Jones	3	aux[11]
a[12]	Robinson	2	Taylor	3	aux[12]
a[13]	Smith	4	Williams	3	aux[13]
a[14]	Taylor	3	Garcia	4	aux[14]
a[15]	Thomas	4	Johnson	4	aux[15]
a[16]	Thompson	4	Smith	4	aux[16]
a[17]	White	2	Thomas	4	aux[17]
a[18]	Williams	3	Thompson	4	aux[18]
a[19]	Wilson	4	Wilson	4	aux[19]

Radix Sorts – LSD and MSD

Radix sorts

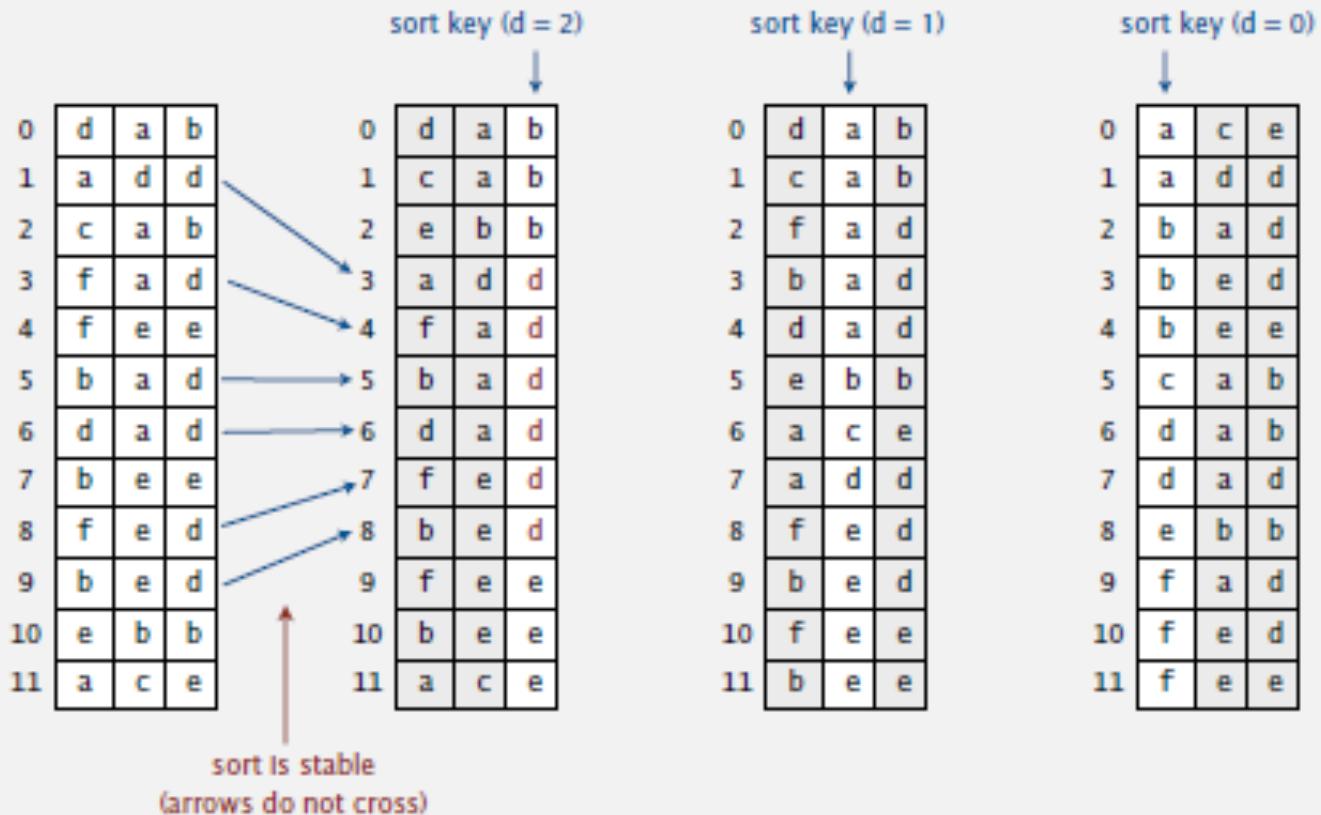
- › Non-comparative integer sorting algorithm
- › Sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.
- › LSD radix sort
 - short keys come before longer keys
 - keys of the same length are sorted lexicographically
 - i.e., normal order of integer representations, such as the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
- › MSD radix sort
 - lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations
 - A sequence such as "b, c, d, e, f, g, h, i, j, ba" would be lexicographically sorted as "b, ba, c, d, e, f, g, h, i, j"
 - (1 to 10 would be output as 1, 10, 2, 3, 4, 5, 6, 7, 8, 9)

LSD Sort

LSD Sort – Sort by Least Significant Digit first

LSD string (radix) sort.

- Consider characters from right to left.
 - Stably sort using d^{th} character as the key (using key-indexed counting).



LSD exercise

Provide the trace of sorting the array of strings given in a table below using LSD sort, providing an equivalent table for each of the 3 passes of the algorithm.

A	R	M
C	A	T
C	A	B
A	R	T
B	U	G
B	U	S

LSD stability and correctness

LSD string sort: correctness proof

Proposition. LSD sorts fixed-length strings in ascending order.

Pf. [by induction on i]

After pass i , strings are sorted by last i characters.

- If two strings differ on sort key,
key-indexed sort puts them in proper
relative order.
- If two strings agree on sort key,
stability keeps them in proper relative order.

sort key ↓

0	d	a	b	0	a	c	e
1	c	a	b	1	a	d	d
2	f	a	d	2	b	a	d
3	b	a	d	3	b	e	d
4	d	a	d	4	b	e	e
5	e	b	b	5	c	a	b
6	a	c	e	6	d	a	b
7	a	d	d	7	d	a	d
8	f	e	d	8	e	b	b
9	b	e	d	9	f	a	d
10	f	e	e	10	f	e	d
11	b	e	e	11	f	e	e

sorted from
previous passes
(by Induction)

Proposition. LSD sort is stable.

Pf. Key-indexed counting is stable.

LSD in Java

```
public class LSD
{
    public static void sort(String[] a, int W) ← fixed-length W strings
    {
        int R = 256; ← radix R
        int N = a.length;
        String[] aux = new String[N];

        for (int d = W-1; d >= 0; d--) ← do key-indexed counting
        {                                for each digit from right to left
            int[] count = new int[R+1];
            for (int i = 0; i < N; i++)
                count[a[i].charAt(d) + 1]++;
            for (int r = 0; r < R; r++)
                count[r+1] += count[r];
            for (int i = 0; i < N; i++)
                aux[count[a[i].charAt(d)]++] = a[i];
            for (int i = 0; i < N; i++)
                a[i] = aux[i];
        }
    }
}
```

Performance wrt other sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	<code>compareTo()</code>
mergesort	$N \lg N$	$N \lg N$	N	✓	<code>compareTo()</code>
quicksort	$1.39 N \lg N^*$	$1.39 N \lg N$	$c \lg N$		<code>compareTo()</code>
heapsort	$2 N \lg N$	$2 N \lg N$	1		<code>compareTo()</code>
LSD sort †	$2 W(N + R)$	$2 W(N + R)$	$N + R$	✓	<code>charAt()</code>

* probabilistic

† fixed-length W keys

LSD Performance

- › Key indexed counting
 - $11n+4R+1$
 - › Initialize arrays $n+R+1$
 - › First loop $3n$
 - › Second loop $3R$
 - › Third loop $5n$
 - › Fourth loop $2n$
- › LSD
 - $10wn+n+WR$
 - $W \times LSD$ apart from third loop which is just $1 \times$ rather than $LSD \times$
 - R usually much smaller than N , so proportional to wn – linear!

Example - sorting large integer arrays

String sorting interview question

Problem. Sort one million 32-bit integers.

Ex. Google (or presidential) interview.

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD string sort.

LSD Sorting large integer arrays

- › <https://algs4.cs.princeton.edu/51radix/LSD.java.html>
- › Break up 32-bit integer in 4 8-bit characters
- › Use bit shifting and masking to isolate characters to sort by
- › What if strings are not same length? MSD!

MSD Sort

MSD – Sort by Most Significant Digit first

› Does simply reversing LSD work?

Reverse LSD

- Consider characters from left to right.
- Stably sort using d^{th} character as the key (using key-indexed counting).

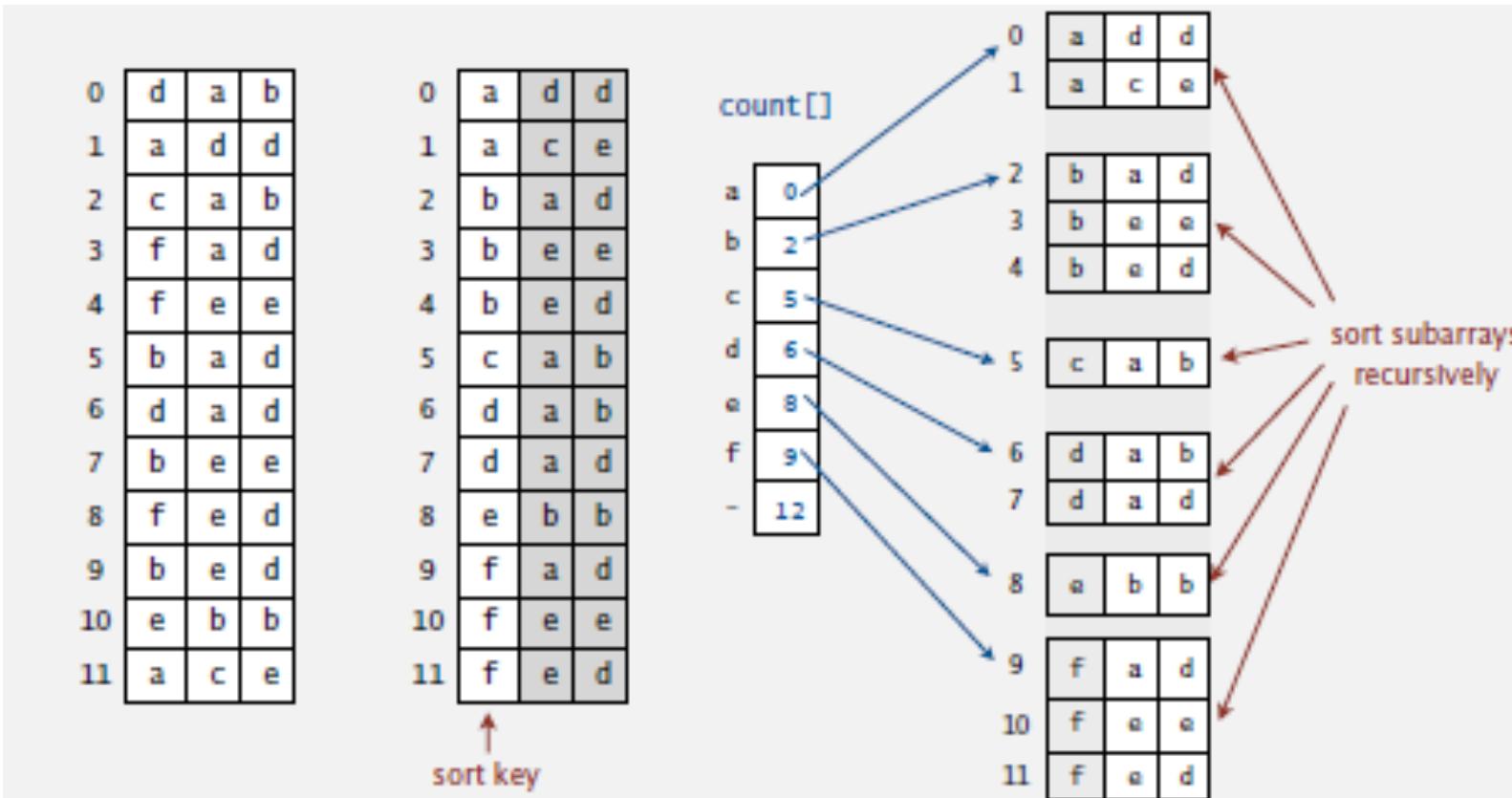
	sort key (d = 0)			sort key (d = 1)			sort key (d = 2)		
0	d	a	b	a	d	d	b	a	d
1	a	d	d	a	c	e	c	a	b
2	c	a	b	b	a	d	d	a	b
3	f	a	d	b	e	e	f	a	d
4	f	e	e	b	e	d	e	b	b
5	b	a	d	c	a	b	a	c	e
6	d	a	d	d	a	b	a	d	d
7	b	e	e	d	a	d	b	e	d
8	f	e	d	e	b	b	b	e	d
9	b	e	d	f	a	d	b	c	e
10	e	b	b	f	e	e	f	e	d
11	a	c	e	f	e	d	b	e	e

not sorted!

MSD string sort

- › Similar to quicksort
- › Partition array into R (radix) pieces according to the first character (most significant digit) using key-indexed counting
- › Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort)

MSD String sort



MSD string sort: example

input	are							
she	are							
sells	by							
seashells	she	sells	seashells	sea	seashells	seashells	seashells	seashells
by	sells	seashells	sea	seashells	seashells	seashells	seashells	seashells
the	seashells	sea	seashells	seashells	seashells	seashells	seashells	seashells
sea	sea	sells						
shore	shore	seashells	sells	sells	sells	sells	sells	sells
the	shells	she						
shells	she	shore						
she	sells	shells						
sells	surely	she						
are	seashells	surely						
surely	the							
seashells	the							

need to examine
every character
in equal keys

| are |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| by |
| sea |
| seashells |
| seashells |
| sells |
| sells |
| she |
shore	shore	shore	shells	she	she	she	she	she
shells	shells	shells	she	shells	shells	shells	shells	shells
she	she	she	shore	shore	shore	shore	shore	shore
surely								
the								
the								

end of string
goes before any
char value

output

Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted)

Variable length Strings

Treat strings as if they had an extra char at end (smaller than any char).

```
private static int charAt(String s, int d)
{
    if (d < s.length()) return s.charAt(d);
    else return -1;
}
```

C strings. Have extra char '\0' at end \Rightarrow no extra work needed.

MSD String sort – Java Implementation

```
public static void sort(String[] a)
{
    aux = new String[a.length]; ←
    sort(a, aux, 0, a.length - 1, 0);
}

private static void sort(String[] a, String[] aux, int lo, int hi, int d)
{
    if (hi <= lo) return;
    int[] count = new int[R+2]; ← key-indexed counting
    for (int i = lo; i <= hi; i++)
        count[charAt(a[i], d) + 2]++;
    for (int r = 0; r < R+1; r++)
        count[r+1] += count[r];
    for (int i = lo; i <= hi; i++)
        aux[count[charAt(a[i], d) + 1]++] = a[i];
    for (int i = lo; i <= hi; i++)
        a[i] = aux[i - lo];

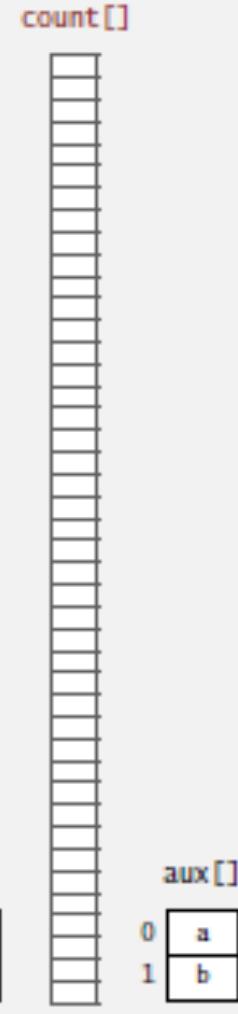
    for (int r = 0; r < R; r++) ← sort R subarrays recursively
        sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}
```

MSD – improvements

Observation 1. Much too slow for small subarrays.

- Each function call needs its own `count[]` array.
- ASCII (256 counts): 100x slower than copy pass for $N=2$.
- Unicode (65,536 counts): 32,000x slower for $N=2$.

Observation 2. Huge number of small subarrays
because of recursion.



Yep, you guessed it – cutoff to Insertion sort

Cutoff to insertion sort

Solution. Cutoff to insertion sort for small subarrays.

- Insertion sort, but start at d^{th} character.

```
private static void sort(String[] a, int lo, int hi, int d)
{
    for (int i = lo; i <= hi; i++)
        for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
            exch(a, j, j-1);
}
```

- Implement `less()` so that it compares starting at d^{th} character.

```
private static boolean less(String v, String w, int d)
{
    for (int i = d; i < Math.min(v.length(), w.length()); i++)
    {
        if (v.charAt(i) < w.charAt(i)) return true;
        if (v.charAt(i) > w.charAt(i)) return false;
    }
    return v.length() < w.length();
}
```

MSD Performance

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✓	compareTo()
quicksort	$1.39 N \lg N^*$	$1.39 N \lg N$	$c \lg N$		compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1		compareTo()
LSD sort †	$2 W(N+R)$	$2 W(N+R)$	$N+R$	✓	charAt()
MSD sort ‡	$2 W(N+R)$	$N \log_R N$	$N + D R$	✓	charAt()

D = function-call stack depth
(length of longest prefix match)

* probabilistic
† fixed-length W keys
‡ average-length W keys

MSD improvements - American flag sort

- › Analogy to Dutch national flag, partition array into many “stripes”
- › In-place variant of radix sort that distributes items into hundreds of buckets
- › How?
- › Cut off to insertion sort
- › Replaces recursion with explicit stack
- › In-place, eliminate auxiliary array (no stability)

MSD vs Quicksort for String sorting

Disadvantages of MSD string sort.

- Extra space for aux[].
 - Extra space for count[].
 - Inner loop has a lot of instructions.
 - Accesses memory "randomly" (cache inefficient).

Disadvantage of quicksort.

- Linearithmic number of string compares (not linear).
 - Has to rescan many characters in keys with long prefix matches.

Goal. Combine advantages of MSD and quicksort.

**doesn't rescan
characters**

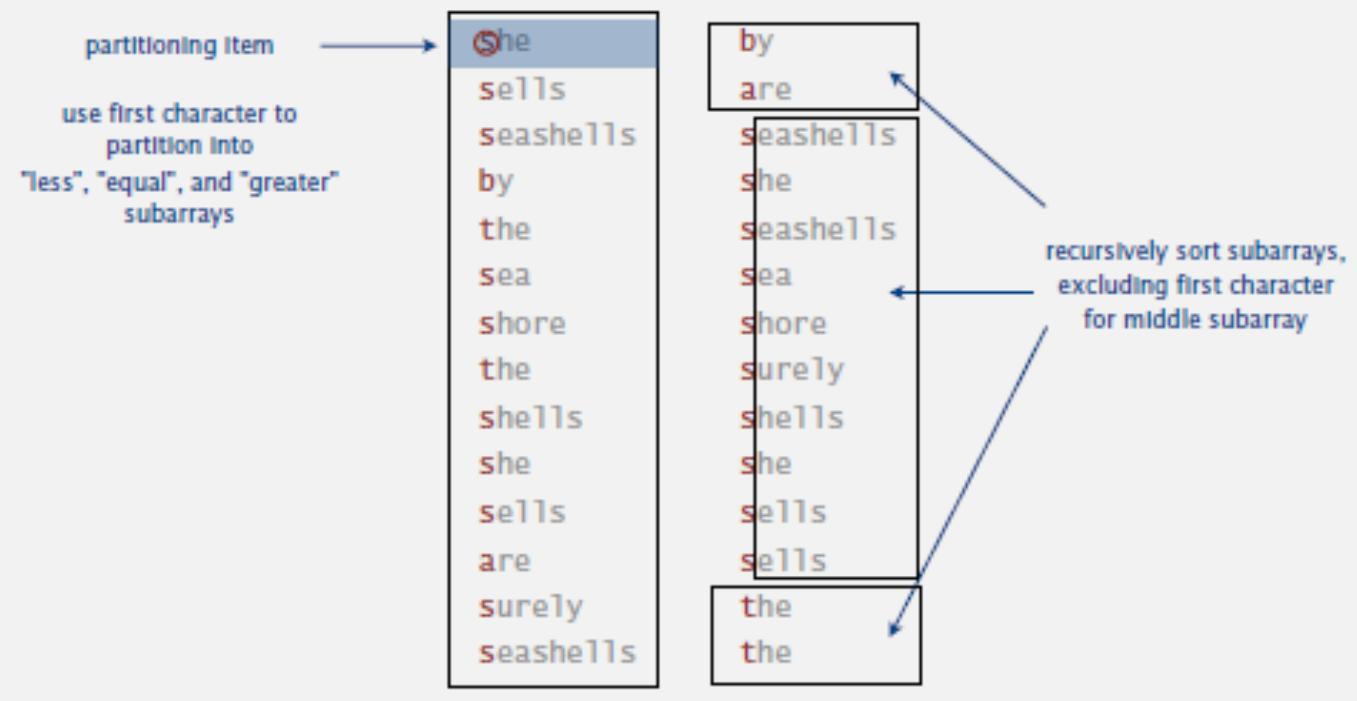
**tight Inner loop,
cache friendly**

› 3-way string quicksort – addressed inefficiency of both

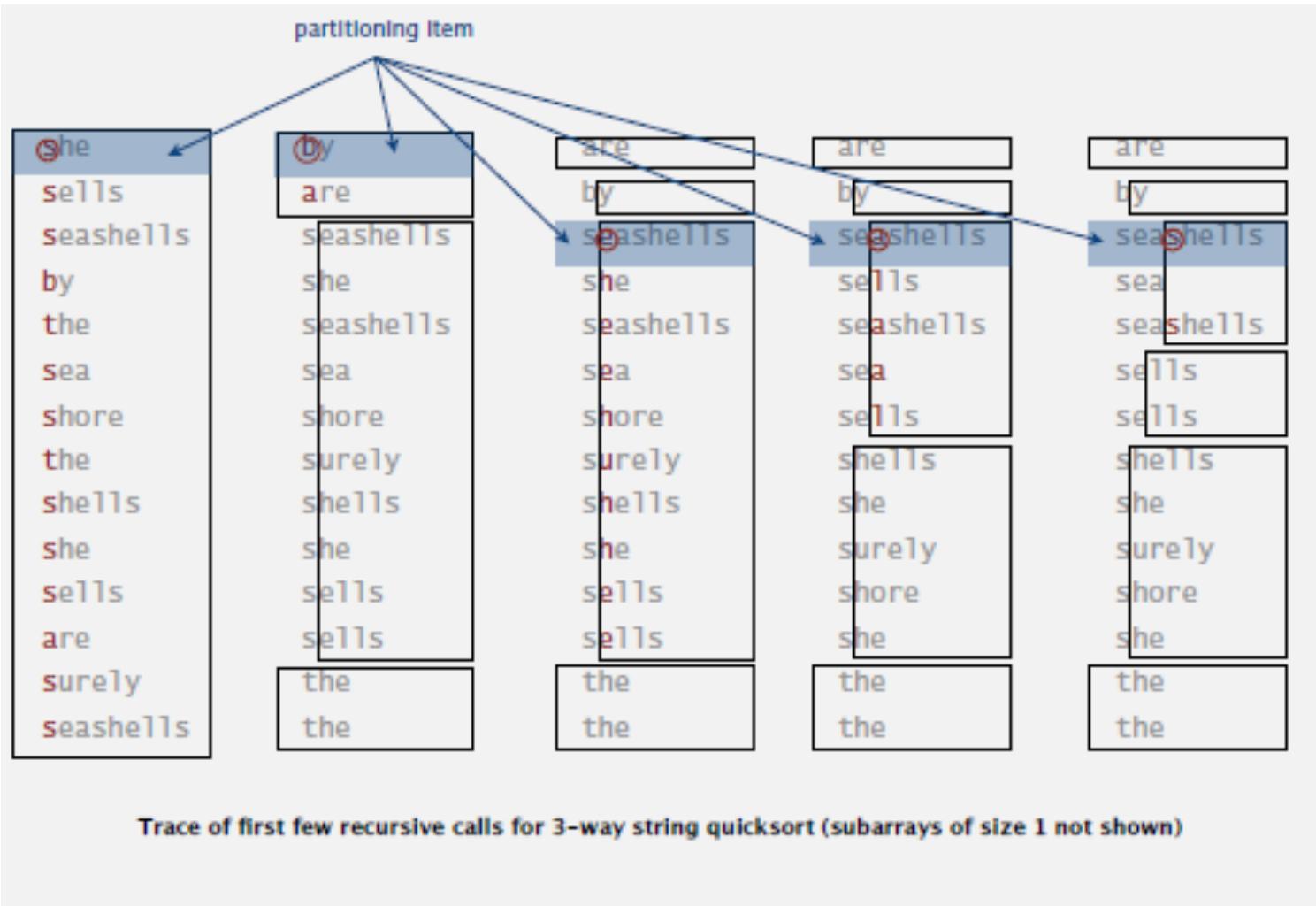
3-way string quicksort

Overview. Do 3-way partitioning on the d^{th} character.

- Less overhead than R -way partitioning in MSD string sort.
- Does not re-examine characters equal to the partitioning char.
(but does re-examine characters not equal to the partitioning char)



3-way string quicksort



3-way string quicksort in java

- › `charAt` instead of `compare`

```
private static void sort(String[] a)
{ sort(a, 0, a.length - 1, 0); }

private static void sort(String[] a, int lo, int hi, int d)
{
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    int v = charAt(a[lo], d); 3-way partitioning  
(using dth character)
    int i = lo + 1;
    while (i <= gt)
    {
        int t = charAt(a[i], d);
        if      (t < v) exch(a, lt++, i++);
        else if (t > v) exch(a, i, gt--);
        else            i++;
    }

    sort(a, lo, lt-1, d);
    if (v >= 0) sort(a, lt, gt, d+1); sort 3 subarrays recursively
    sort(a, gt+1, hi, d);
}
```

Summary of string sorts

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✓	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✓	compareTo()
quicksort	$1.39 N \lg N^*$	$1.39 N \lg N$	$c \lg N$		compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1		compareTo()
LSD sort †	$2 W(N+R)$	$2 W(N+R)$	$N+R$	✓	charAt()
MSD sort †	$2 W(N+R)$	$N \log_R N$	$N+DR$	✓	charAt()
3-way string quicksort	$1.39 WN \lg R^*$	$1.39 N \lg N$	$\log N + W$		charAt()

String sorting algorithms – when to use which?

- › Insertion
 - Small arrays, arrays in (almost)order
- › Quick
 - General purpose when space is tight
- › Merge
 - General purpose stable
- › 3-way quick
 - Large number of equal keys
- › LSD
 - Short fixed-length strings
- › MSD
 - Random strings
- › 3-way string quicksort
 - General purpose, strings with long pre-fix matches

Check if 2 strings are anagrams

- › two words are anagrams if they contain the same characters
 - “abc” and “cba” are anagrams

CS22012: Data Structures and Algorithms II

Substring Search

Ivana.Dusparic@scss.tcd.ie

Outline of Substring search algorithms

- › Brute force
 - › KMP (Knuth-Morris-Pratt)
 - › Boyer-Moore
 - › Rabin-Karp
 - › Many many many others
-
- › Suffix arrays
 - › LCP (longest common prefix) arrays

String matching algorithms

- › [Handbook of Exact String Matching Algorithms](#)
- › <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.133.4896&rep=rep1&type=pdf>
- › <http://www-igm.univ-mly.fr/~lecroq/string/>

Java String implementation

[Java library](#). The `indexOf()` method in Java's string library returns the index of the first occurrence of a given string, starting at a given offset.

- › Which algorithm does `String.indexOf(String)` use?
 - Naïve loop (brute force)
 - Why?
- › `String.contains()`

Common interview questions

- › Implement a needle-in-a-haystack
 - public int Search(String haystack, String needle)
- › Implement strstr()
 - Find the first instance of a string in another string
- › Longest common substring between 2 files
- › Longest substring that's a palindrome
- › Longest repeated substring
- › Etc etc

Different to Pattern Matching

- › Find a pattern, i.e. one of the specified set of substrings in a text
- › Regular expression – notation to specify a set of strings
- › For more info see 5.4 in Sedgewick and Wayne

Substring search - definition

Goal. Find pattern of length M in a text of length N .

typically $N \gg M$

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A

↑
match

Substring search – brute force

Check for pattern starting at each text position.

i	j	i+j	0	1	2	3	4	5	6	7	8	9	10
			txt → A	B	A	C	A	D	A	B	R	A	C
0	2	2	A	B	R	A							
1	0	1		A	B	R	A						
2	1	3			A	B	R	A					
3	0	3				A	B	R	A				
4	1	5					A	B	R	A			
5	0	5						A	B	R	A		
6	4	10							A	B	R	A	

Annotations:

- Red arrows point to entries in red: "pat" (pattern) at index 6, "mismatches" at index 3, and "return i when j is M" at index 6.
- Gray arrows point to entries in gray: "entries in black match the text" at index 5, "for reference only" at index 4, and "match" at index 7.

Substring search – brute force

Check for pattern starting at each text position.

i	j	i+j	0	1	2	3	4	5	6	7	8	9	10
			A	B	A	C	A	D	A	B	R	A	C
4	3	7					A	D	A	C	R		
5	0	5					A	D	A	C	R		

```
public static int search(String pat, String txt)
{
    int M = pat.length();
    int N = txt.length();
    for (int i = 0; i <= N - M; i++)
    {
        int j;
        for (j = 0; j < M; j++)
            if (txt.charAt(i+j) != pat.charAt(j))
                break;
        if (j == M) return i; ← index in text where
    }                                pattern starts
    return N; ← not found
}
```

Substring search – brute force

Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.

i	j	i+j	0	1	2	3	4	5	6	7	8	9
			txt → A	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B	← pat				
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						A	A	A	A	B

match

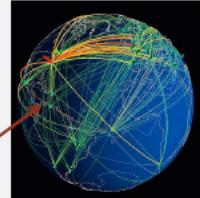
Worst case. $\sim MN$ char compares.

Substring search - backup

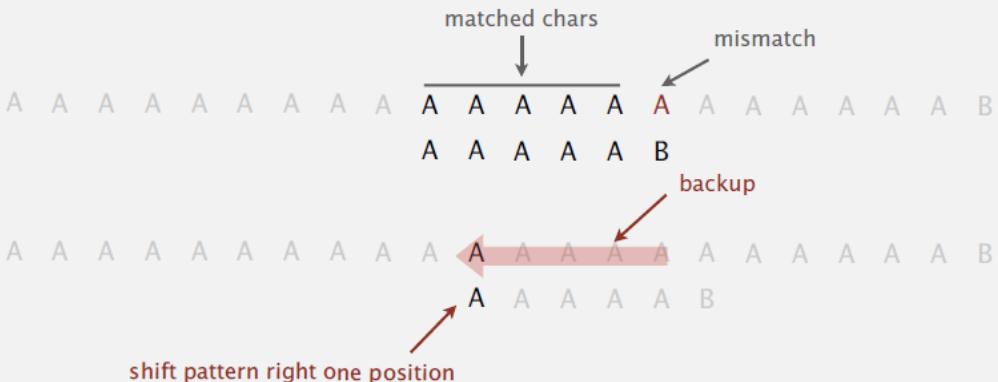
Backup

In many applications, we want to avoid **backup** in text stream.

- Treat input as stream of data.
 - Abstract model: standard input.



Brute-force algorithm needs backup for every mismatch.



Approach 1. Maintain buffer of last M characters.

Approach 2. Stay tuned.

Substring search – explicit backup

Same sequence of char compares as previous implementation.

- i points to end of sequence of already-matched chars in text.
- j stores # of already-matched chars (end of sequence in pattern).

<u>i</u>	<u>j</u>	0	1	2	3	4	5	6	7	8	9	10
		A	B	A	C	A	D	A	B	R	A	C
7	3				A	D	A	C	R			
5	0					A	D	A	C	R		

```
public static int search(String pat, String txt)
{
    int i, N = txt.length();
    int j, M = pat.length();
    for (i = 0, j = 0; i < N && j < M; i++)
    {
        if (txt.charAt(i) == pat.charAt(j)) j++;
        else { i -= j; j = 0; } ← explicit backup
    }
    if (j == M) return i - M;
    else return N;
}
```

Algorithmic challenges in substring search

Brute-force is not always good enough.

Theoretical challenge. Linear-time guarantee. ← fundamental algorithmic problem

Practical challenge. Avoid backup in text stream. ← often no room or time to save text

Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for each good person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for each person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.

Knuth-Morris-Pratt (KMP)

KMP

- › 1970 by Donald Knuth and Vaughan Pratt
- › + Independently by James H. Morris.
- › (Donald Knuth - The Art of Computer Programming - comprehensive monograph that covers many kinds of programming algorithms and their analysis – 4 volumes and counting)

KMP

Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern BAAAAAAAAA.

- Suppose we match 5 chars in pattern, with mismatch on 6th char.
- We know previous 6 chars in text are BAAAAB.
- Don't need to back up text pointer!

assuming { A, B } alphabet

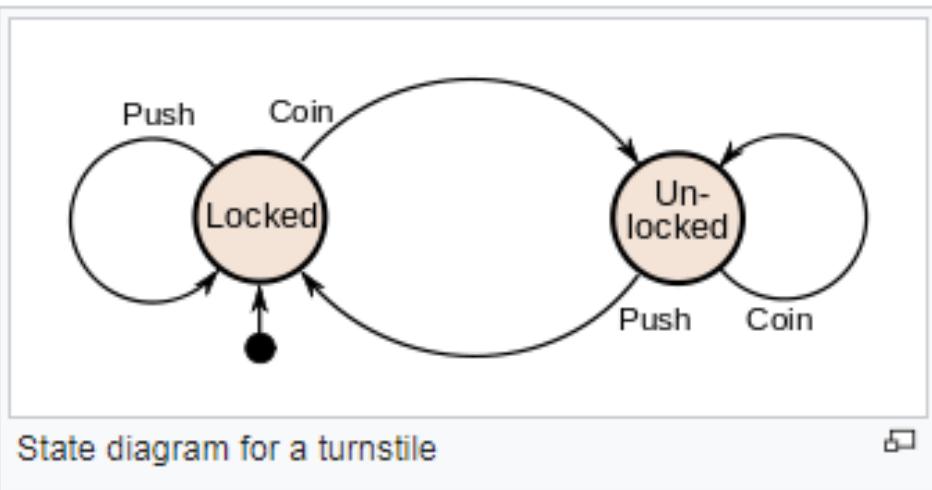


Knuth-Morris-Pratt algorithm. Clever method to always avoid backup. (!)

KMP – avoid back up how? DFA

- › DFA – Deterministic Final State Automaton
- › Finite State Automaton/Finite State Machine
 - mathematical model of computation
 - an abstract machine that can be in exactly one of a finite number of states at any given time.
 - can change from one state to another in response to some external inputs
 - the change from one state to another is called a transition
 - defined by a list of its states, its initial state, and the conditions for each transition.
- › Deterministic - produces a unique computation (or run) of the automaton for each input string
- › DFA - finite-state machine that accepts and rejects strings of symbols

FSA – an example



Finite State Machine – more formally

A **finite state automaton** is a quintuple (Q, Σ, E, S, F) with

- Q a finite set of states
- Σ a finite set of symbols, the alphabet
- $S \subseteq Q$ the set of start states
- $F \subseteq Q$ the set of final states
- E a set of edges $Q \times (\Sigma \cup \{\epsilon\}) \times Q$

The **transition function** d can be defined as

$$d(q, a) = \{q' \in Q | \exists (q, a, q') \in E\}$$

- Deterministic Finite Automata are always *complete*: they define a transition for each state and each input symbol.

DFA

DFA is abstract string-searching machine.

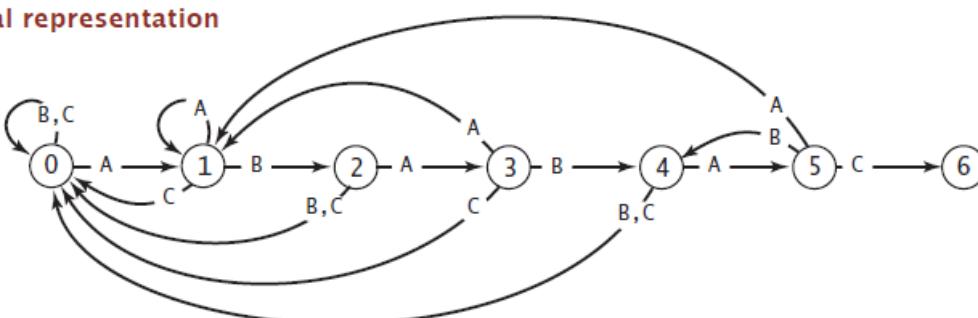
- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state.

internal representation

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	1	1	3	1	5	1
	A	0	2	0	4	0
	B	0	0	4	0	4
	C	0	0	0	0	6

If in state j reading char c:
if j is 6 halt and accept
else move to state dfa[c][j]

graphical representation

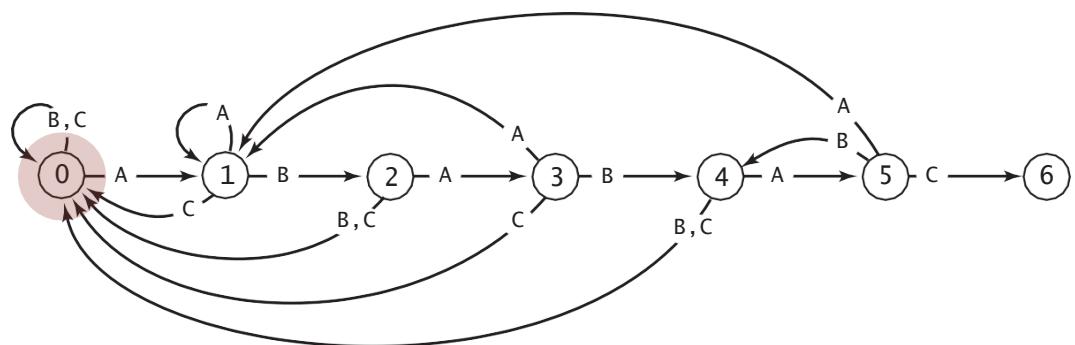


DFA simulation

DFA simulation

B C B A A B A C A A B A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

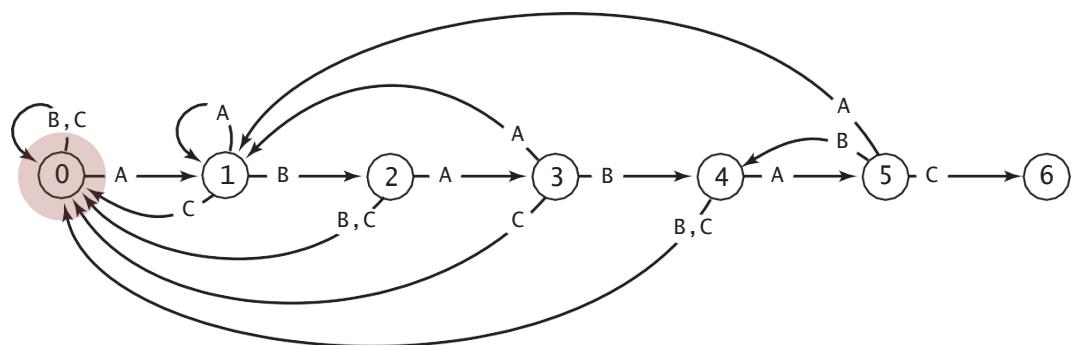


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

B C B A A B A C A A B A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

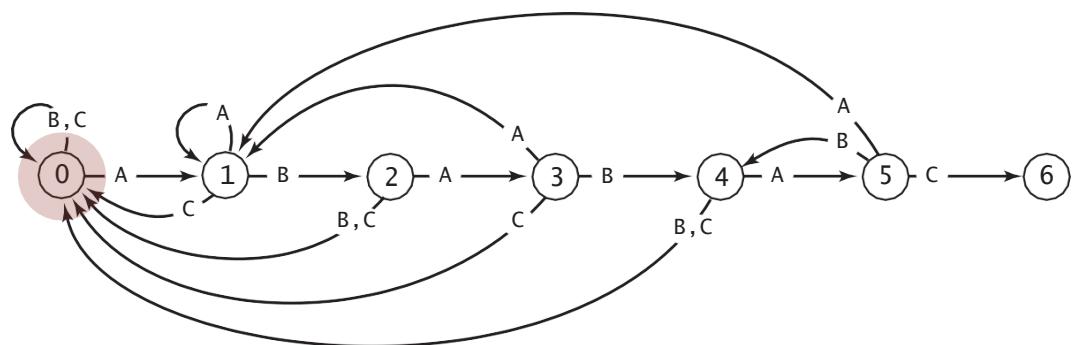


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

B C B A A B A C A A B A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

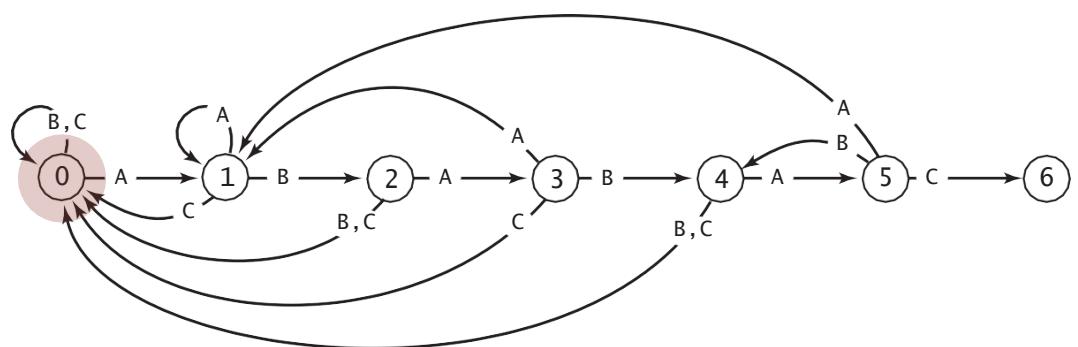


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

B C B A A B A C A A B A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

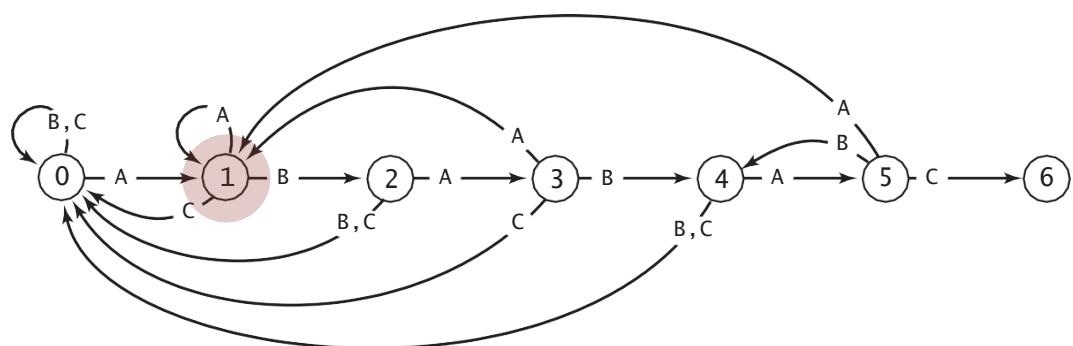


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

B C B A A B A C A A B A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

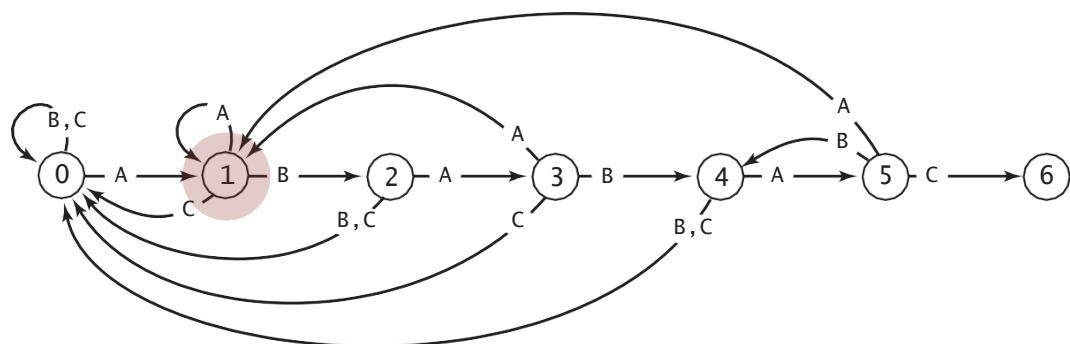


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

B C B A A B A C A A A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

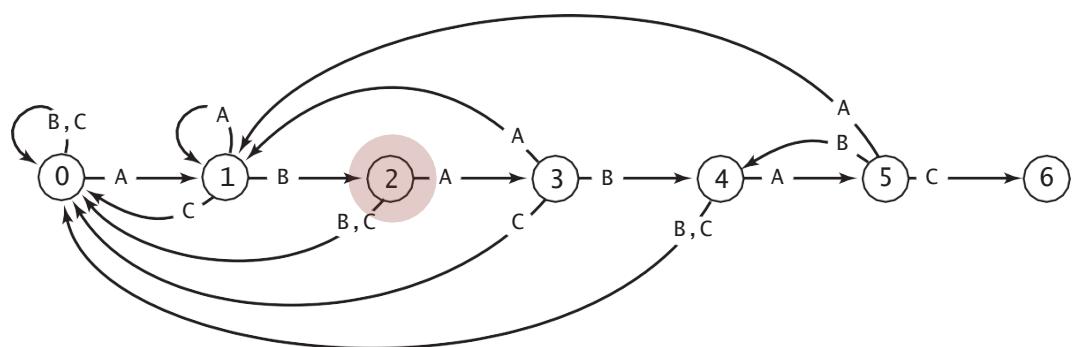


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A B A B A C

DFA simulation

B C B A A B A C A A B A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

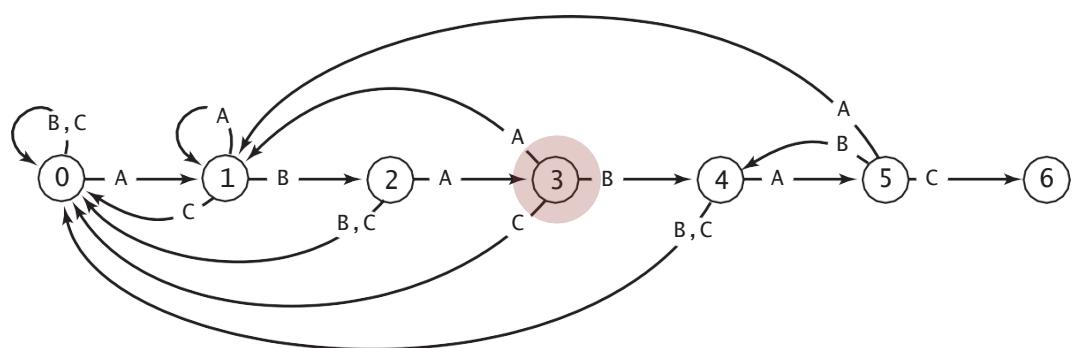


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A
B A B A C

DFA simulation

B C B A A B A C A A B A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

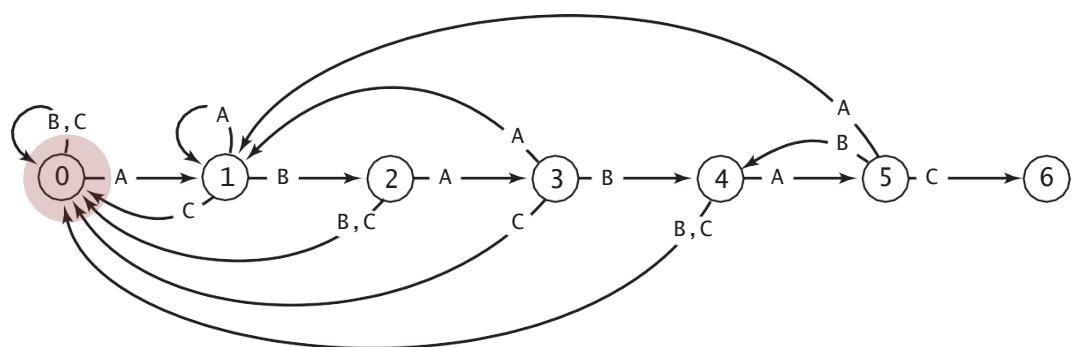


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A
B A B A C

DFA simulation

B C B A A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

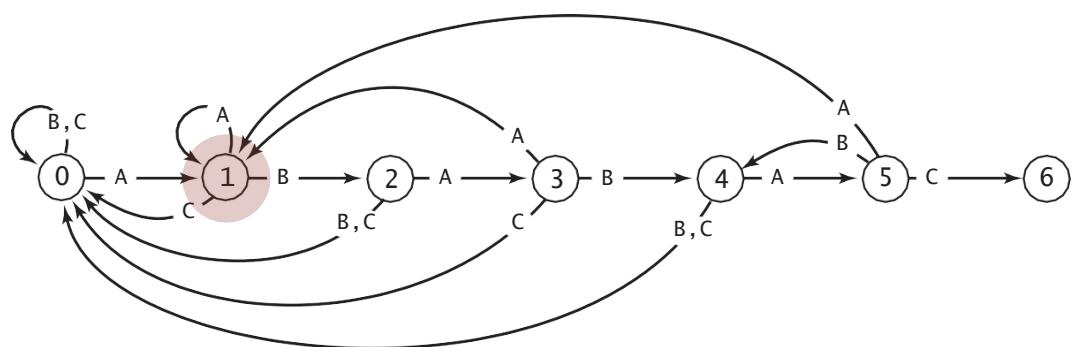


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A
B A B A C

DFA simulation

B C B A A B A C A A
 ↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

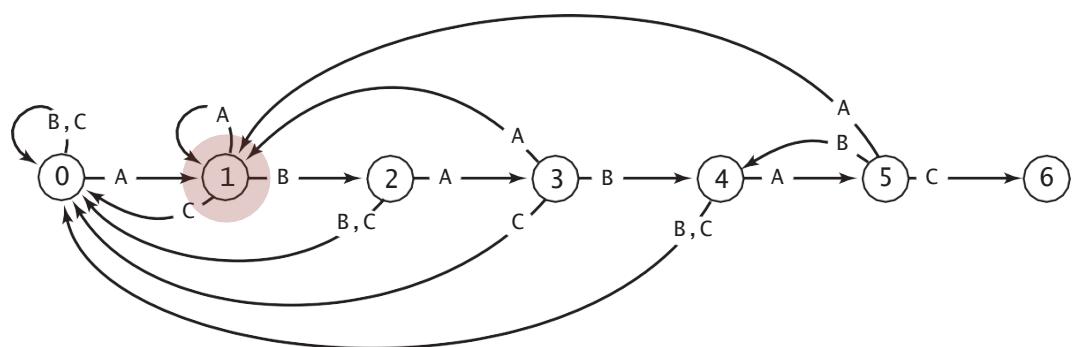


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A
B A B A C

DFA simulation

B C B A A B A C A A
 ↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

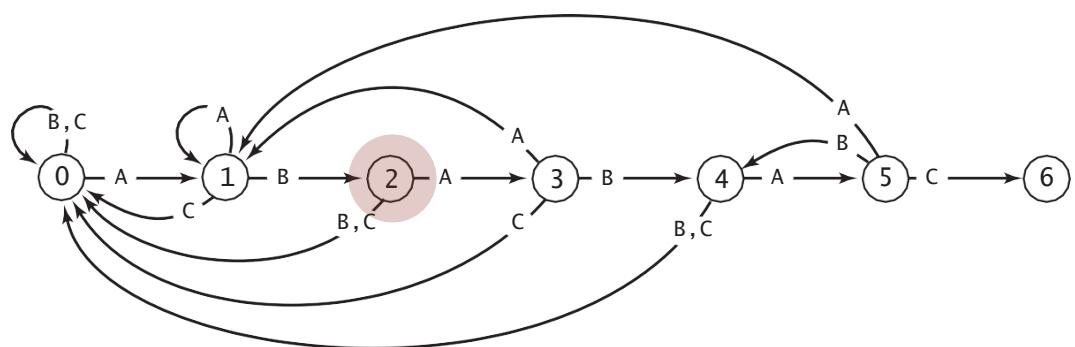


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A
B A B A C

DFA simulation

B C B A A B A C A A
A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

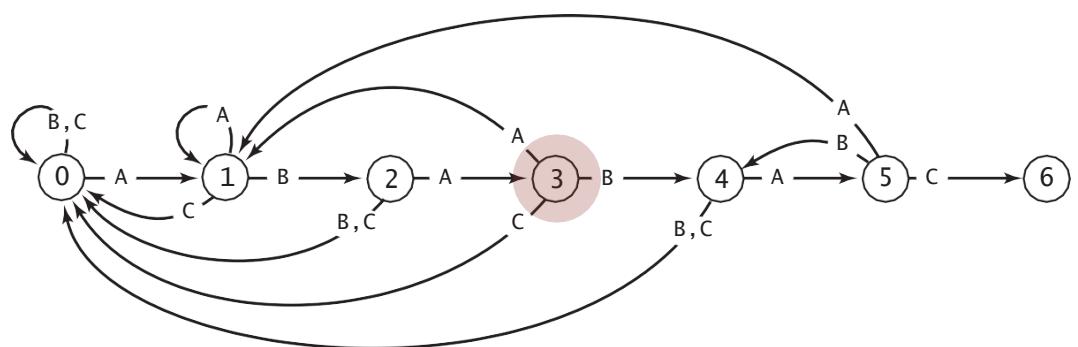


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A
B A B A C

DFA simulation

B C B A A B A C A A B A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

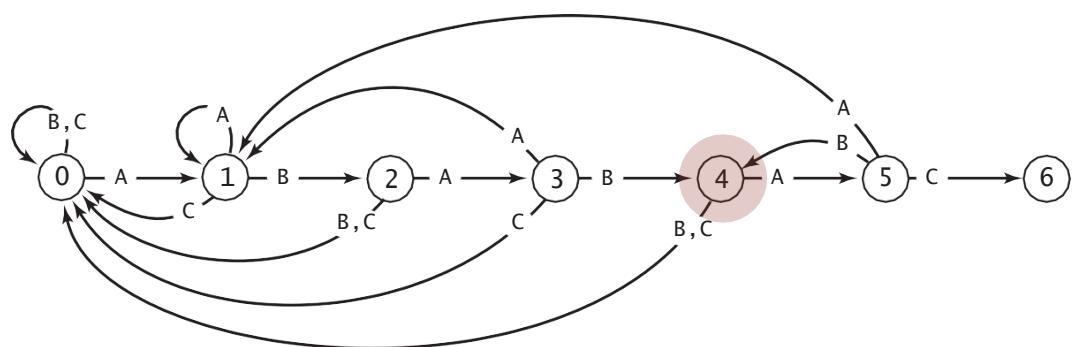


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A
B A B A C

DFA simulation

B C B A A B A C A A B A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

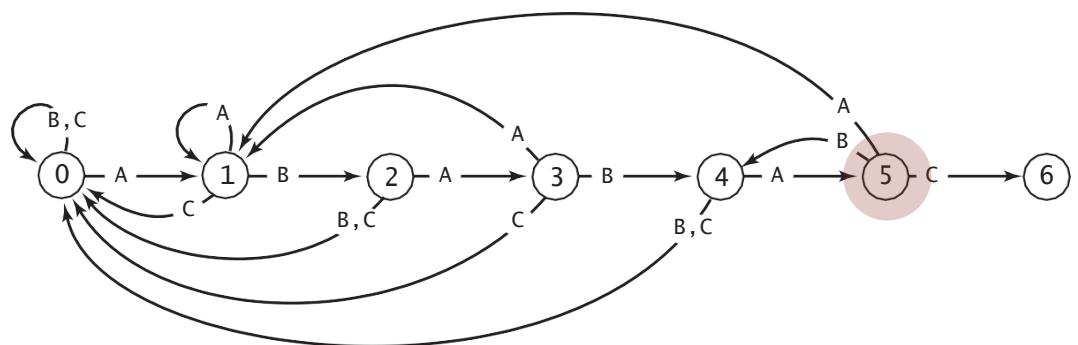


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A
B A B A C

DFA simulation

B C B A A B A C A A B A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

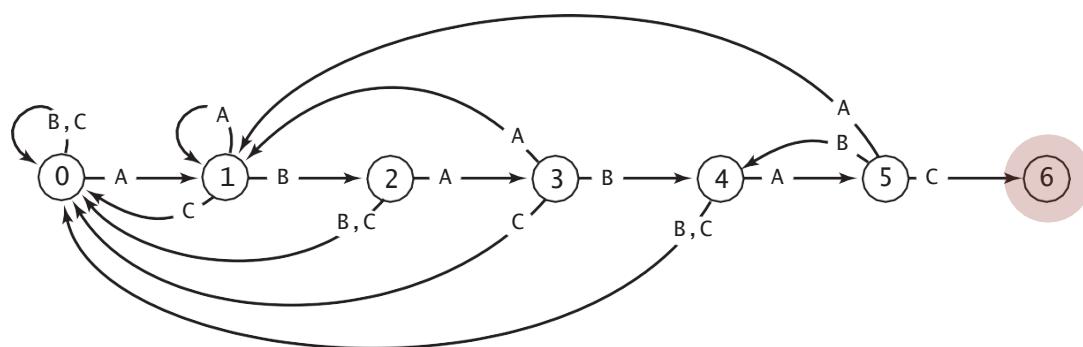


TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A
B A B A C

DFA simulation

B C B A A B A C A A B A B A C A A
↑

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6



TRACE OF KMP SUBSTRING SEARCH (DFA SIMULATION) FOR A
B A B A C

DFA States – number of characters matched

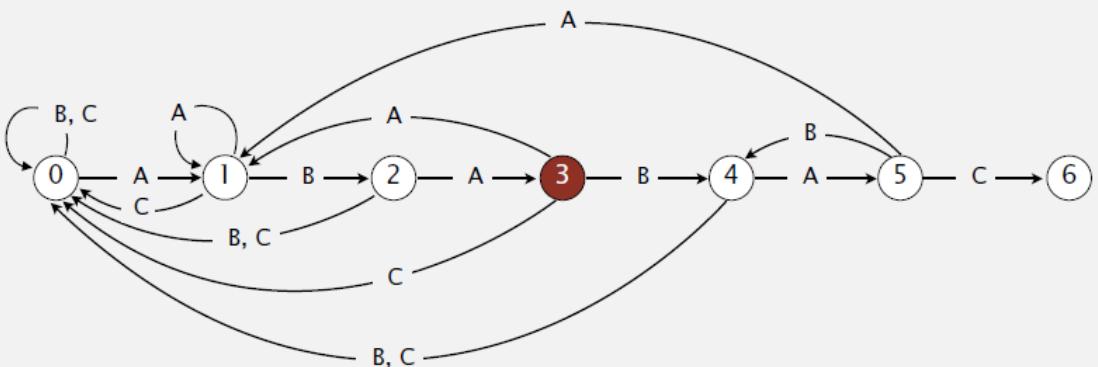
Q. What is interpretation of DFA state after reading in $\text{txt}[i]$?

A. State = number of characters in pattern that have been matched.

length of longest prefix of $\text{pat}[]$
that is a suffix of $\text{txt}[0..i]$

Ex. DFA is in state 3 after reading in $\text{txt}[0..6]$.

i								
0	1	2	3	4	5	6	7	8
B	C	B	A	A	B	A	C	A
suffix of $\text{txt}[0..6]$								
pat								
0	1	2	3	4	5	6	7	8
A	B	A	B	A	C			
prefix of $\text{pat}[]$								



DFA simulation exercise

- › Consider the following DFA for searching for a string “IVANA”
- › For simplicity, we assume the alphabet contains only letters A, I, N, V
- › DFA is therefore as follows

j	0	1	2	3	4
char?	I	V	A	N	A
A	0	0	3	0	5
I	1	1	1	1	1
N	0	0	0	4	0
V	0	2	0	0	0

Exercise 1 Simulating DFA:

- › 1. Construct graphical representation of the DFA table
- › 2. Write the trace of states when searching for a string “IVANA” in input “ANVAIVAAIVANAAN”
- › 3. Vote on turning point for the correct trace

Exercise 1 Simulating DFA:

j	0	1	2	3	4
char?	I	V	A	N	A
A	0	0	3	0	0
I	1	1	1	1	1
N	0	0	0	4	0
V	0	2	0	0	5

Text ANVAIVAAIVANAAN

Search string IVANA

KMP Java Implementation

Key differences from brute-force implementation.

- Need to precompute $\text{dfa}[][]$ from pattern.
- Text pointer i never decrements.

```
public int search(String txt)
{
    int i, j, N = txt.length();
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt.charAt(i)][j];           ← no backup
    if (j == M) return i - M;
    else         return N;
}
```

Running time.

- Simulate DFA on text: at most N character accesses.
- Build DFA: how to do efficiently? [warning: tricky algorithm ahead]

DFA construction

Knuth-Morris-Pratt construction

Include one state for each character in pattern (plus accept state).

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A					

A
B
C



Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Match transition: advance to next state if $c == \text{pat.charAt}(j)$.

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1		3		5	
dfa[][][j]	B		2		4	
C						6

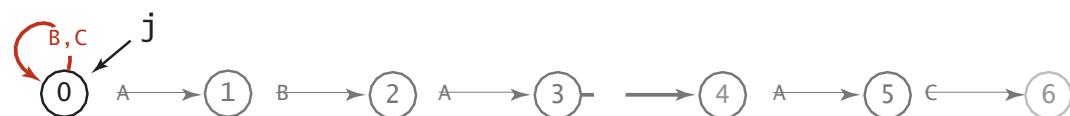


Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1		3		5	
dfa[][][j]	B	0	2		4	
C	0					6

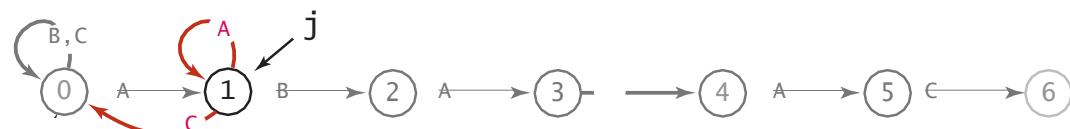


Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3		5	
dfa[][][j]	B	0	2	4		
C	0	0			6	



Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3		5	
dfa[][][j]	B	0	2	0	4	
C	0	0	0			6

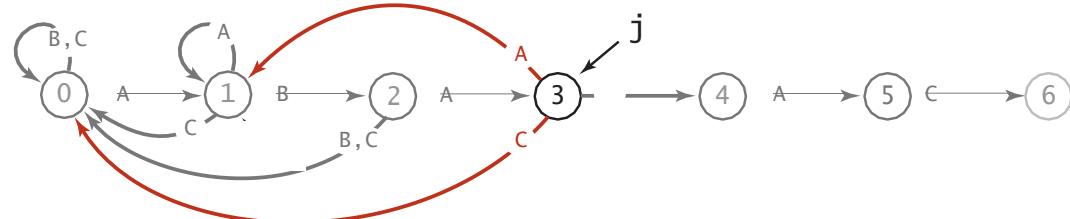


Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	
dfa[][][j]	B	0	2	0	4	
C	0	0	0	0	6	

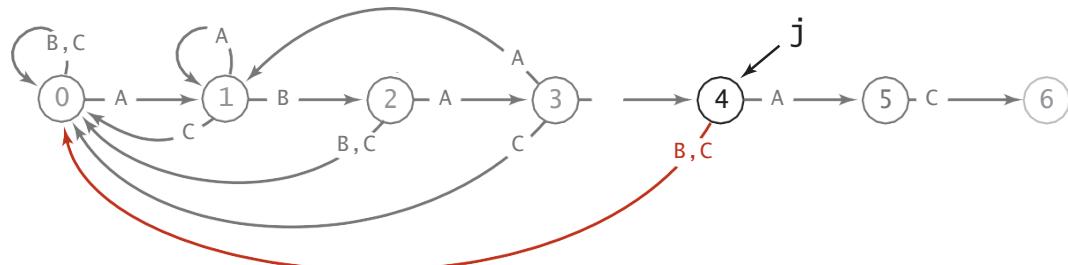


Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

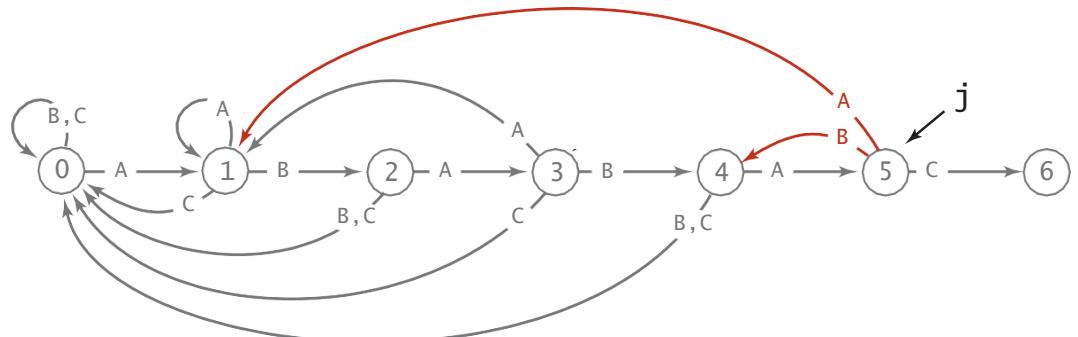
j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6



Knuth-Morris-Pratt construction

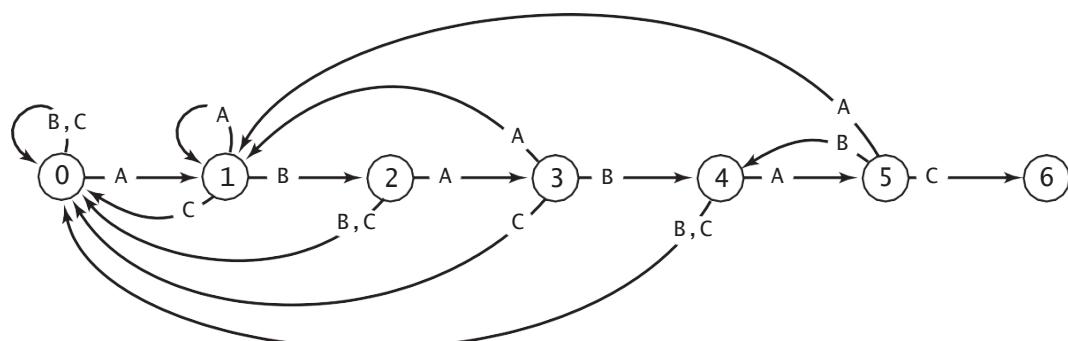
Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
	A	1	1	3	1	5
dfa[][][j]	B	0	2	0	4	0
	C	0	0	0	0	6



Knuth-Morris-Pratt construction

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6



Exercise 2 constructing DFA:

- › Construct DFA table and graphical representation for a search word “banana”
- › Make up a 15-letter string in which you’re going to search for the word, assuming the alphabet contains only letters.
 - You can decide whether you want the string to contain the search word or not, but if it does, do not have it too early into the string
- › Write out the trace of DFA states while searching for the word in the madeup string
- › Hand up the exercise

DFA construction - Java code

Include one state for each character in pattern (plus accept state).

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A	B	C			



Constructing the DFA for KMP substring search for A B A B A C

DFA Construction – Java code

Match transition. If in state j and next char $c == \text{pat.charAt}(j)$, go to $j+1$.

↑
first j characters of pattern
have already been matched ↑
next char matches ↑
now first $j+1$ characters of
pattern have been matched

pat.charAt(j)	0	1	2	3	4	5
A	A	B	A	B	A	C
dfa[] [j]	1		3		5	
C		2		4		6



DFA construction - Java code

Match transition. For each state j , $\text{dfa}[\text{pat.charAt}(j)][j] = j+1$.

↑
first j characters of pattern
have already been matched ↑
now first $j+1$ characters of
pattern have been matched

j	0	1	2	3	4	5
$\text{pat.charAt}(j)$	A	B	A	B	A	C
A	1		3		5	
$\text{dfa}[][][j]$	B		2		4	
C						6



Constructing the DFA for KMP substring search for A B A B A C

DFA Construction – Java code

Mismatch transition. If in state j and next char $c \neq \text{pat.charAt}(j)$, then the last $j-1$ characters of input are $\text{pat}[1..j-1]$, followed by c .

To compute $\text{dfa}[c][j]$: Simulate $\text{pat}[1..j-1]$ on DFA and take transition c .
Running time. Seems to require j steps.

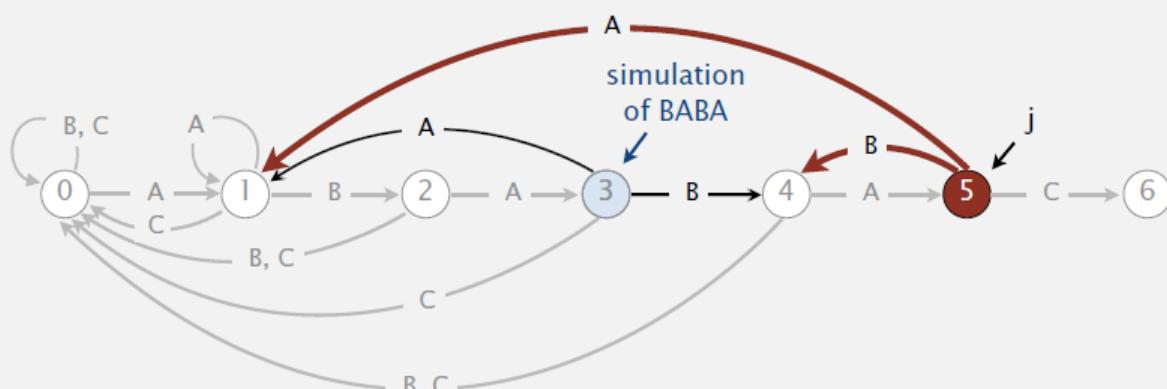
still under construction (!)

Ex. $\text{dfa}['A'][5] = 1; \quad \text{dfa}['B'][5] = 4$

simulate BABA;
take transition 'A'
 $= \text{dfa}['A'][3]$

simulate BABA;
take transition 'B'
 $= \text{dfa}['B'][3]$

$\begin{array}{r|cccccc} j & 0 & 1 & 2 & 3 & 4 & 5 \\ \text{pat.charAt}(j) & A & B & A & B & A & C \end{array}$



DFA construction

- › So lets do this again, while maintaining state x

Knuth-Morris-Pratt construction

Include one state for each character in pattern (plus accept state).

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A					
	B					
	C					



Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Match transition. For each state j , $\text{dfa}[\text{pat.charAt}(j)][j] = j+1$.

↑
first j characters of pattern
have already been matched ↑
now first $j+1$ characters of
pattern have been matched

j	0	1	2	3	4	5
$\text{pat.charAt}(j)$	A	B	A	B	A	C
A	1		3		5	
$\text{dfa}[][][j]$	B		2		4	
C						6



Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Mismatch transition.

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1		3		5	
dfa[][][j]	B	0	2		4	
C	0					6

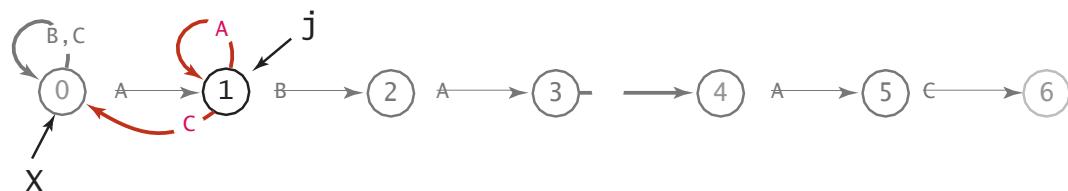


Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$,
 $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

	X	0	1	2	3	4	5
j							
pat.charAt(j)	A	B	A	B	A	C	
A	1	1	3		5		
dfa[] [j]	B	0	2		4		
C	0	0				6	

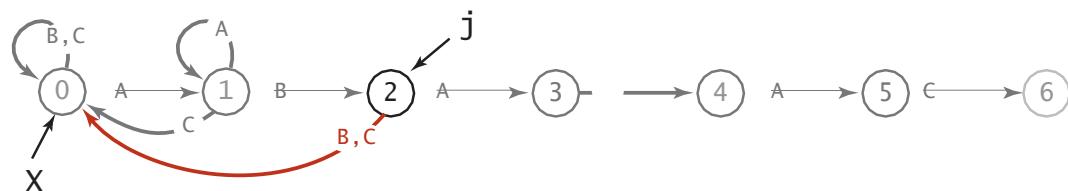


Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$,
 $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

	X	0	1	2	3	4	5
j							
pat.charAt(j)	A	B	A	B	A	C	
A	1	3		5			
dfa[] [j]	B	0	2	0	4		
C	0	0	0			6	

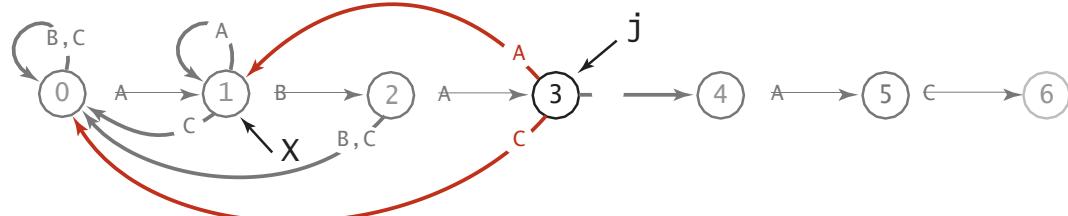


Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$,
 $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

j	0	1	2	3	4	5
$\text{pat.charAt}(j)$	A	B	A	B	A	C
A	1	1	3	1	5	
$\text{dfa}[] [j]$	B	0	2	0	4	
C	0	0	0	0	6	

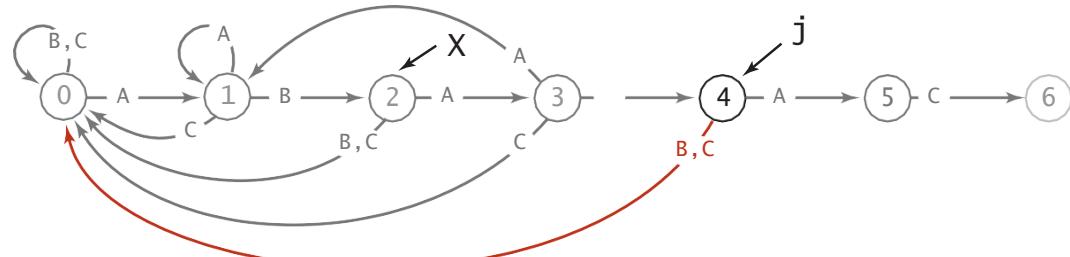


Constructing the DFA for KMP substring search for A B A B A C

Knuth-Morris-Pratt construction

Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$,
 $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

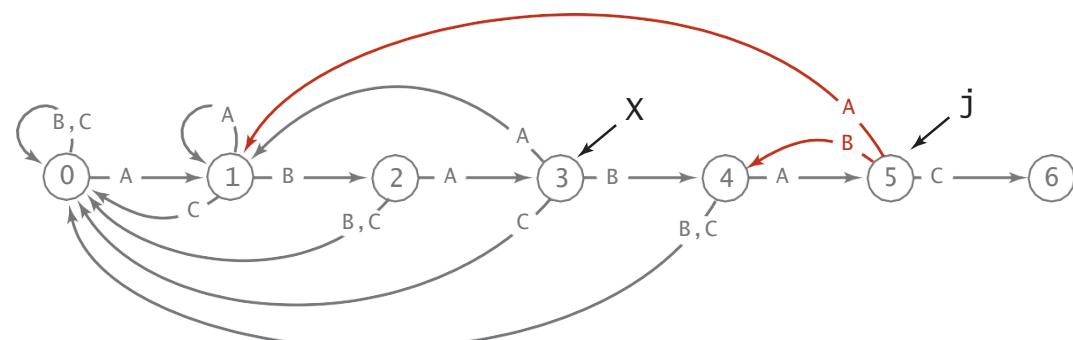
j	0	1	2	3	4	5
$\text{pat.charAt}(j)$	A	B	A	B	A	C
A	1	1	3	1	5	
$\text{dfa}[] [j]$	B	0	2	0	4	0
C	0	0	0	0	0	6



Knuth-Morris-Pratt construction

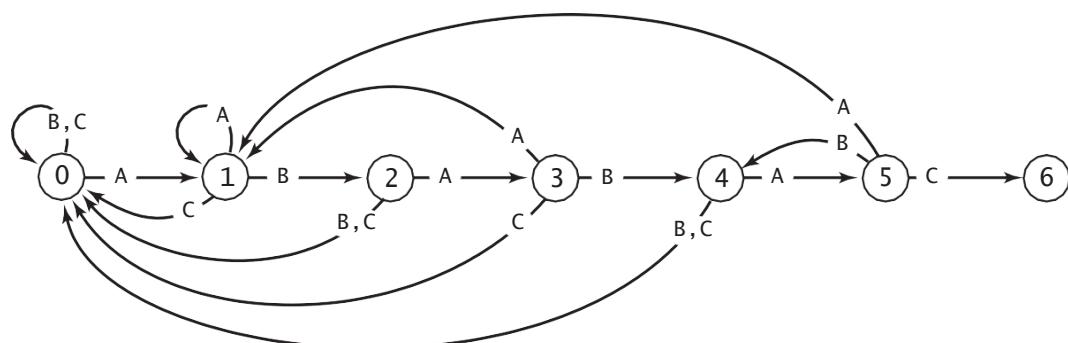
Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$,
 $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

j	0	1	2	3	4	5
$\text{pat.charAt}(j)$	A	B	A	B	A	C
A	1	1	3	1	5	1
$\text{dfa}[] [j]$	B	0	2	0	4	0
C	0	0	0	0	0	6



Knuth-Morris-Pratt construction

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6



DFA Construction – Java code

For each state j :

- Copy $\text{dfa}[][\text{X}]$ to $\text{dfa}[][\text{j}]$ for mismatch case.
- Set $\text{dfa}[\text{pat.charAt(j)}][\text{j}]$ to $\text{j}+1$ for match case.
- Update X .

```
public KMP(String pat)
{
    this.pat = pat;
    M = pat.length();
    dfa = new int[R][M];
    dfa[pat.charAt(0)][0] = 1;
    for (int X = 0, j = 1; j < M; j++)
    {
        for (int c = 0; c < R; c++)
            dfa[c][j] = dfa[c][X]; ← copy mismatch cases
        dfa[pat.charAt(j)][j] = j+1; ← set match case
        X = dfa[pat.charAt(j)][X]; ← update restart state
    }
}
```

Running time. M character accesses (but space/time proportional to $R M$).

KMP search – Java code

```
for (i = 0, j = 0; i < n && j < m; i++) {
    j = dfa[txt.charAt(i)][j];
}
if (j == m) return i - m;      // found
return n;                      // not found
```

KMP search performance

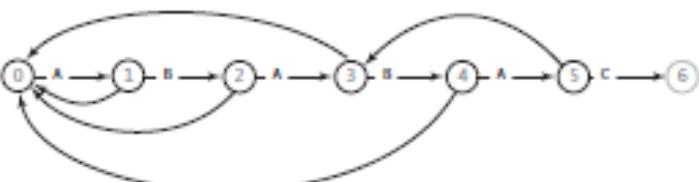
KMP substring search analysis

Proposition. KMP substring search accesses no more than $M+N$ chars to search for a pattern of length M in a text of length N .

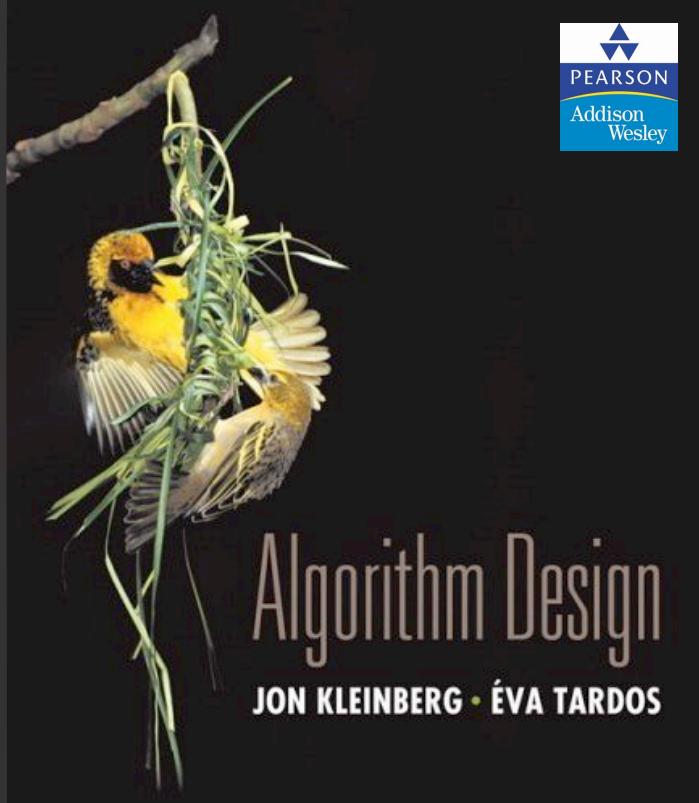
Pf. Each pattern char accessed once when constructing the DFA; each text char accessed once (in the worst case) when simulating the DFA.

Proposition. KMP constructs `dfa[][]` in time and space proportional to $R M$.

Larger alphabets. Improved version of KMP constructs `nfa[]` in time and space proportional to M .



KMP NFA for ABABAC



7. FORD-FULKERSON DEMO

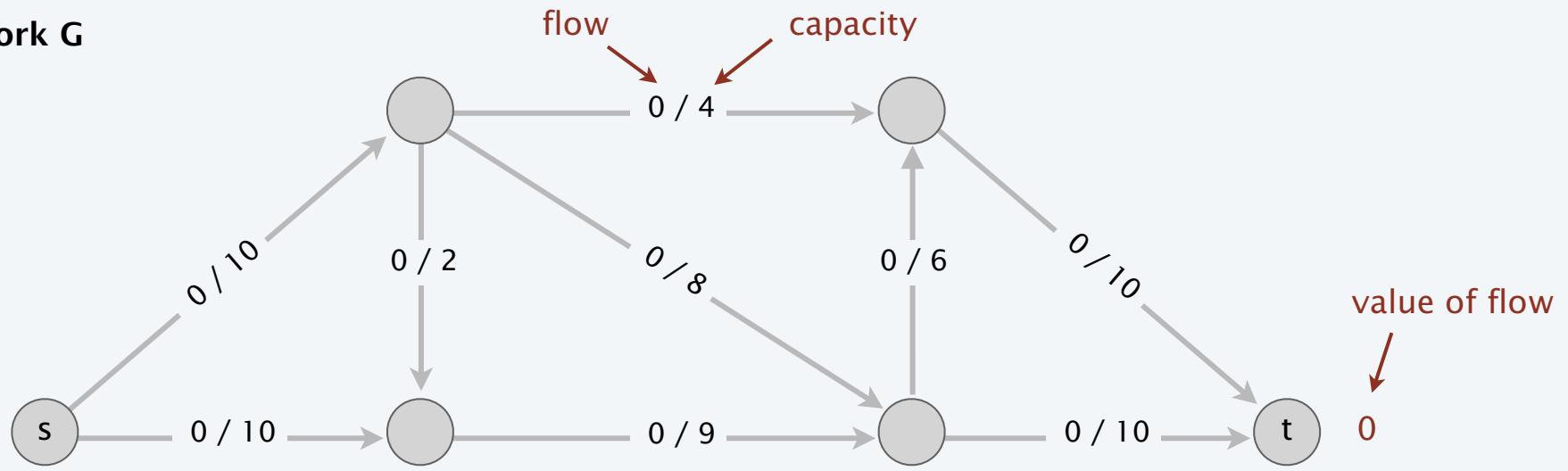
Lecture slides by Kevin Wayne

Copyright © 2005 Pearson–Addison Wesley

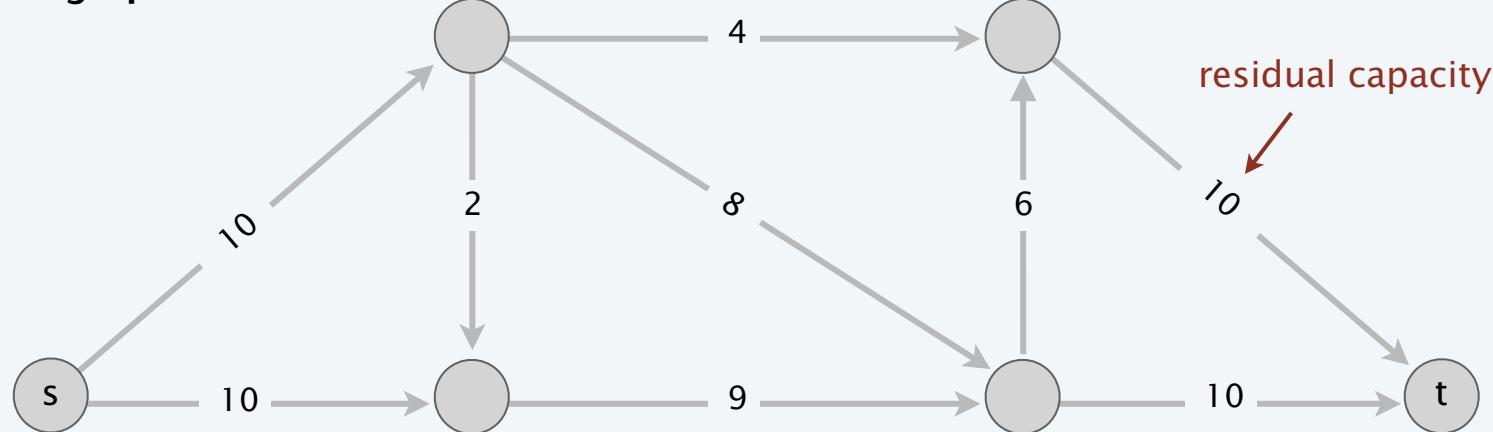
<http://www.cs.princeton.edu/~wayne/kleinberg-tardos>

Ford-Fulkerson algorithm demo

network G

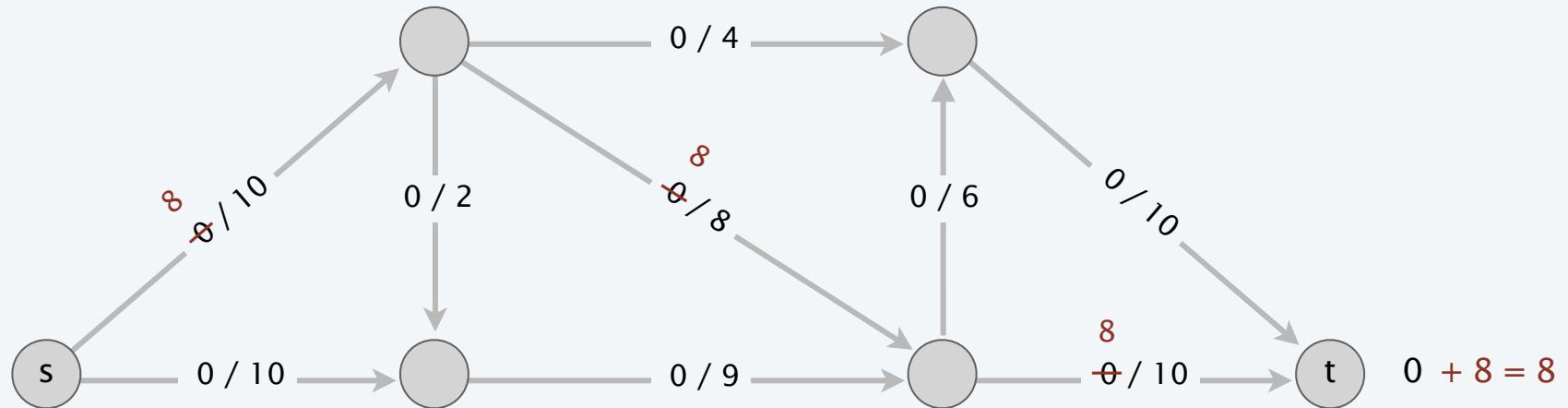


residual graph G_f

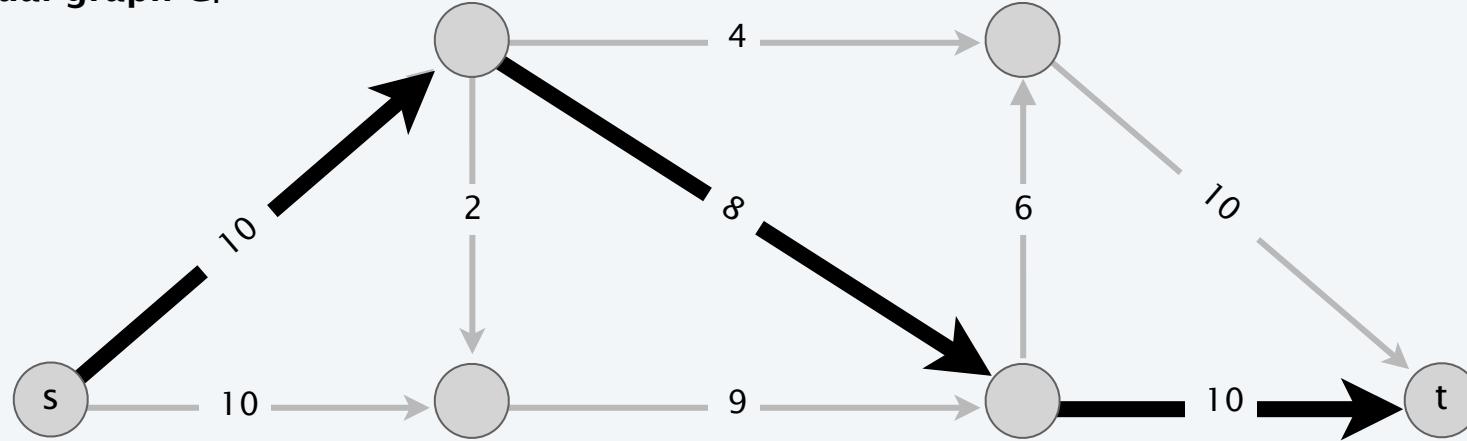


Ford-Fulkerson algorithm demo

network G

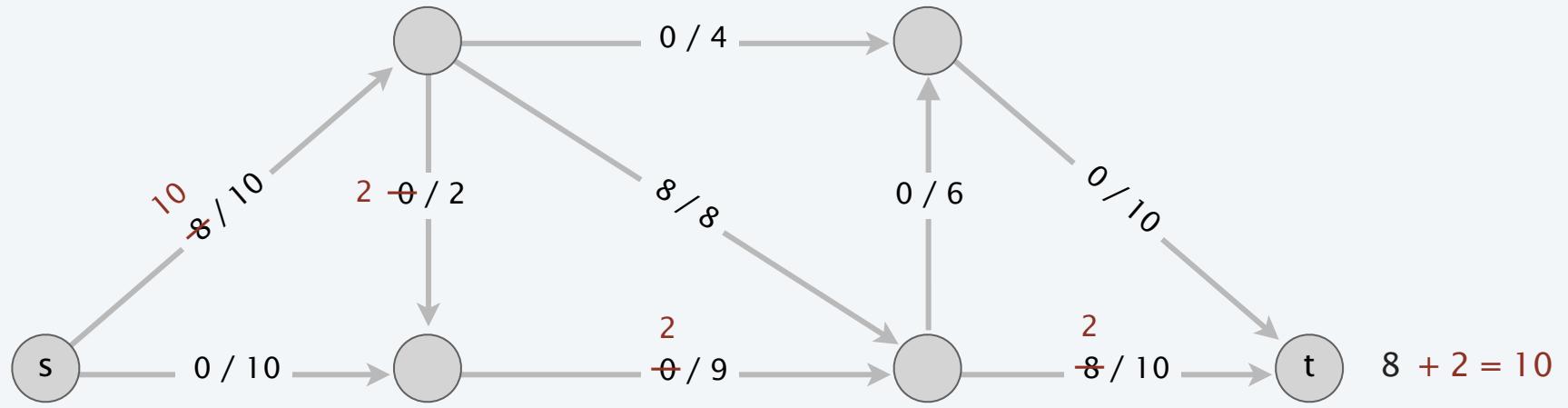


residual graph G_f

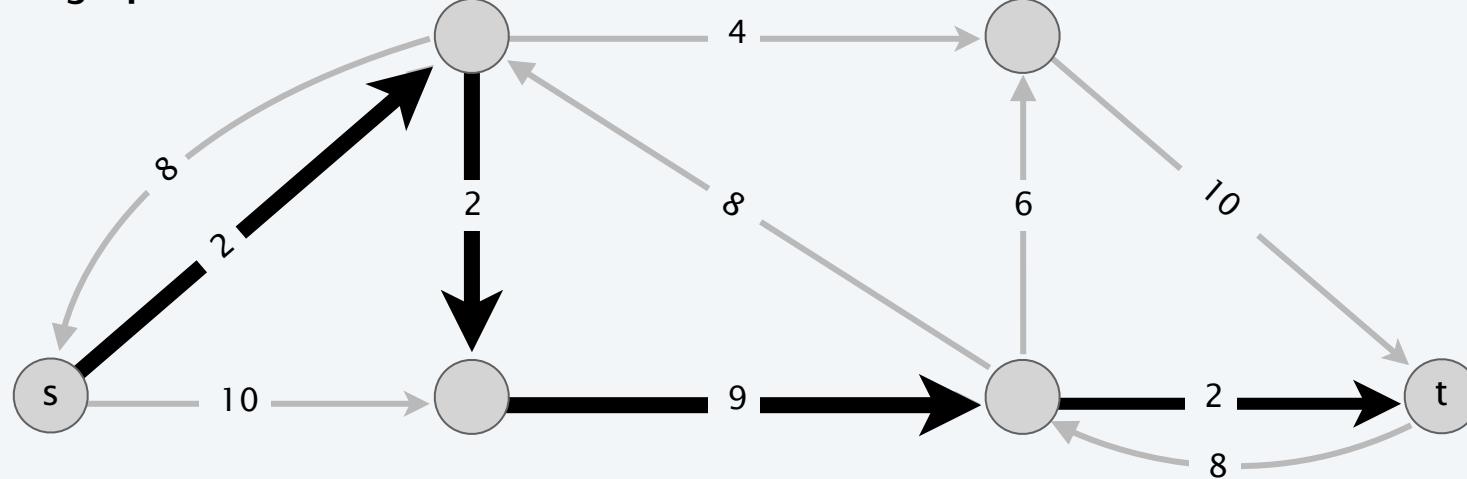


Ford-Fulkerson algorithm demo

network G

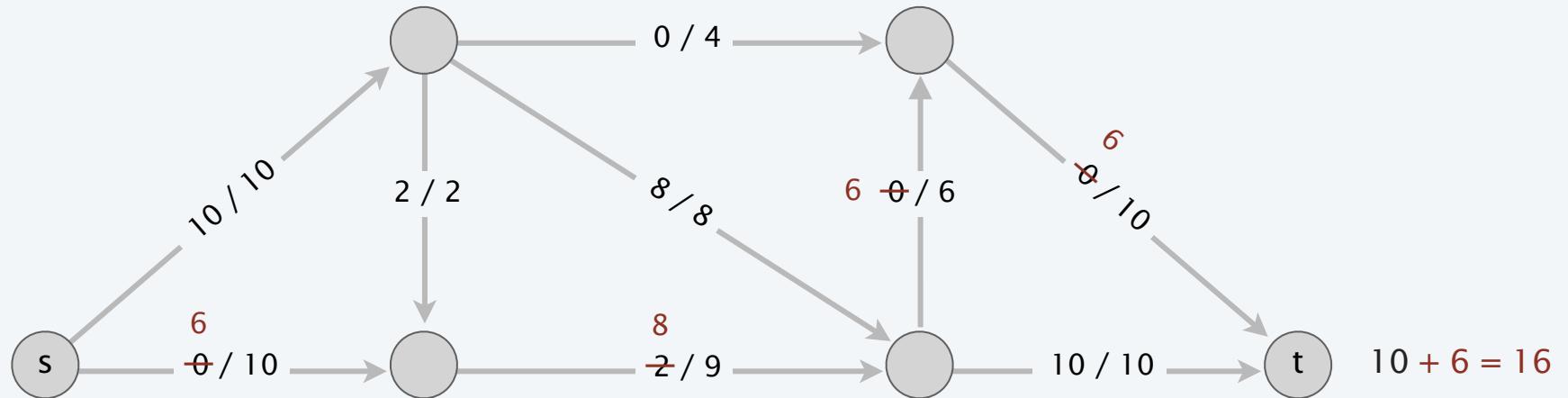


residual graph G_f

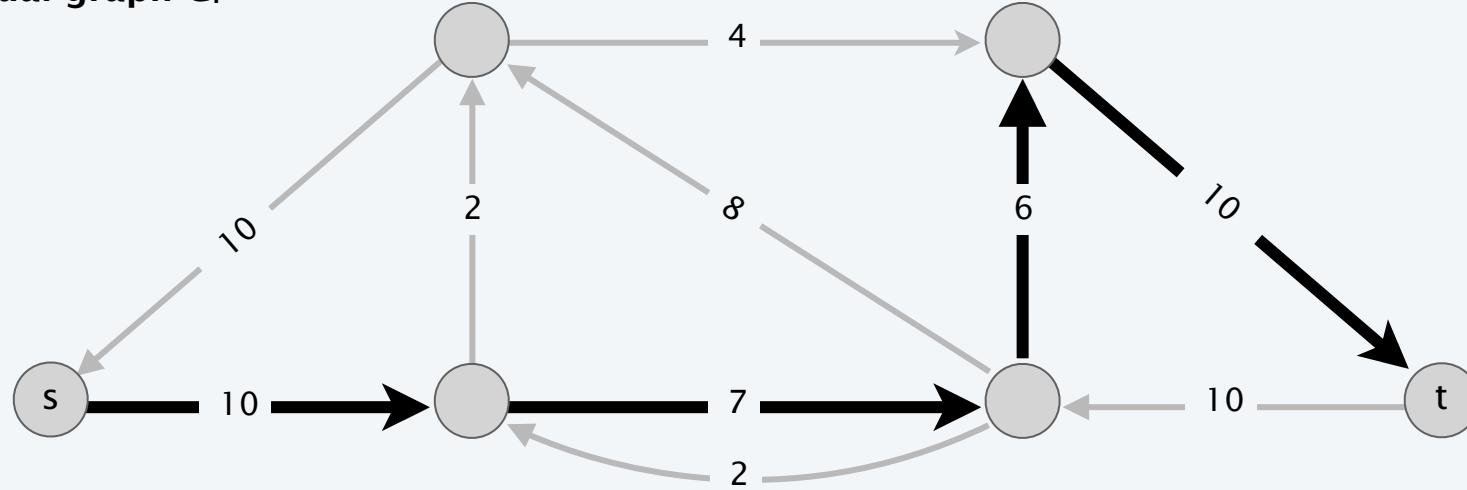


Ford-Fulkerson algorithm demo

network G

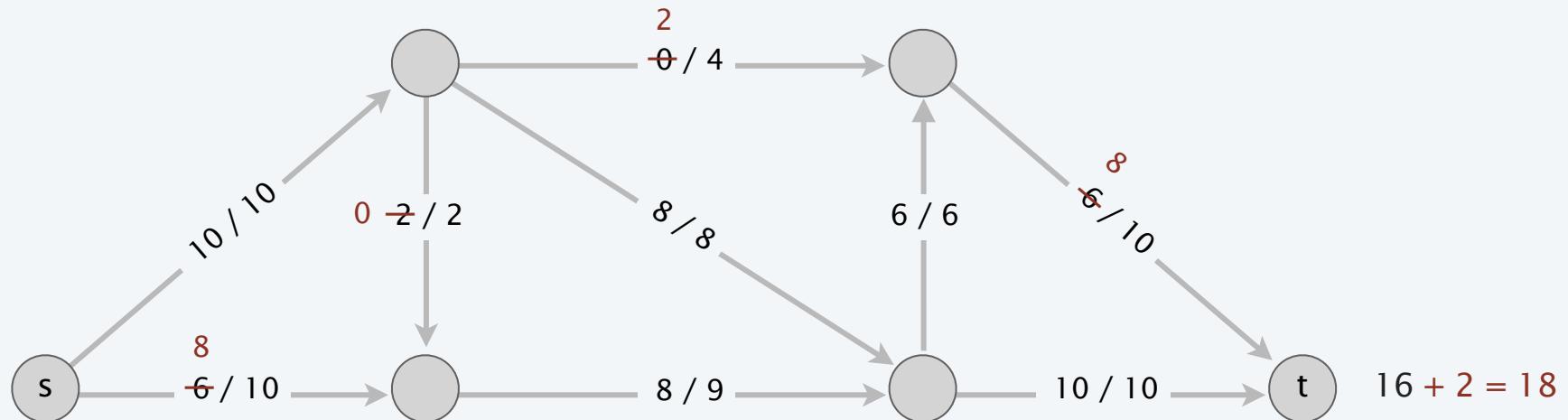


residual graph G_f

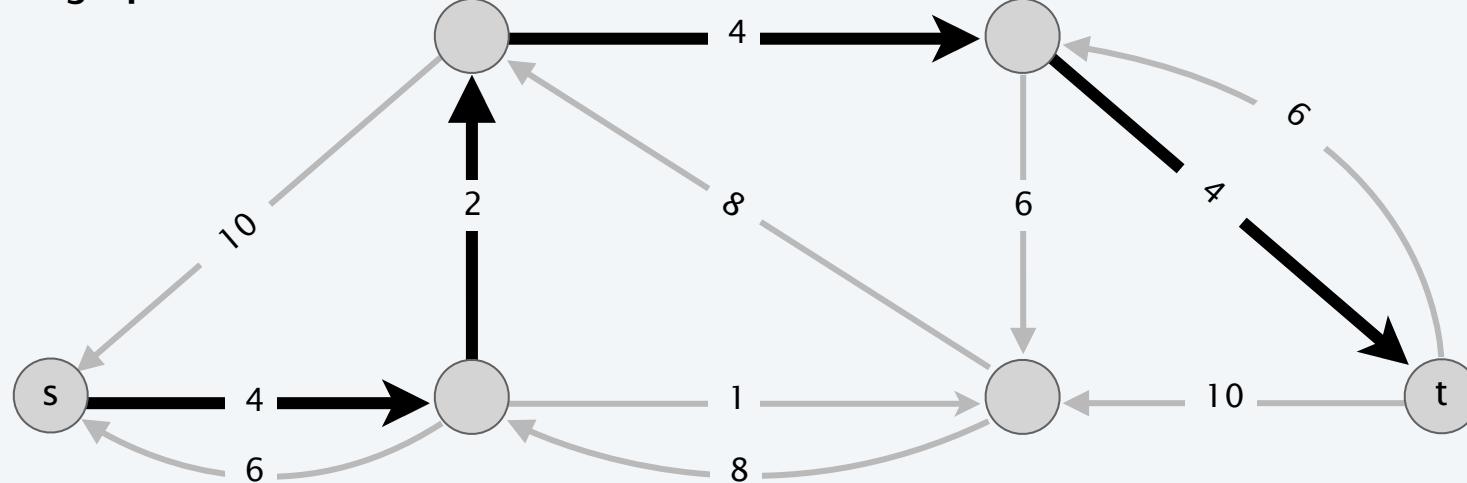


Ford-Fulkerson algorithm demo

network G

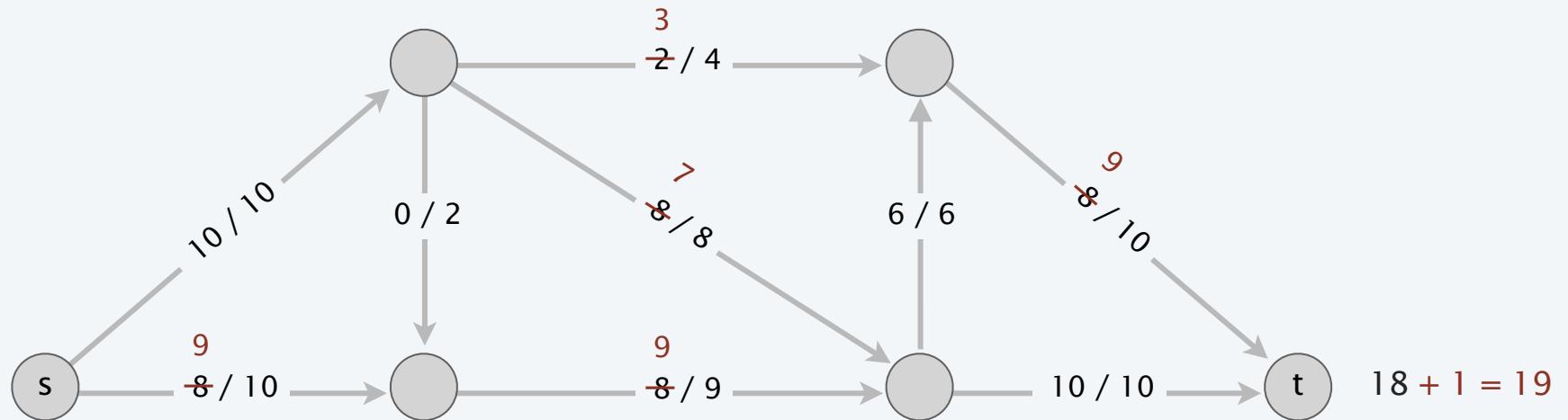


residual graph G_f

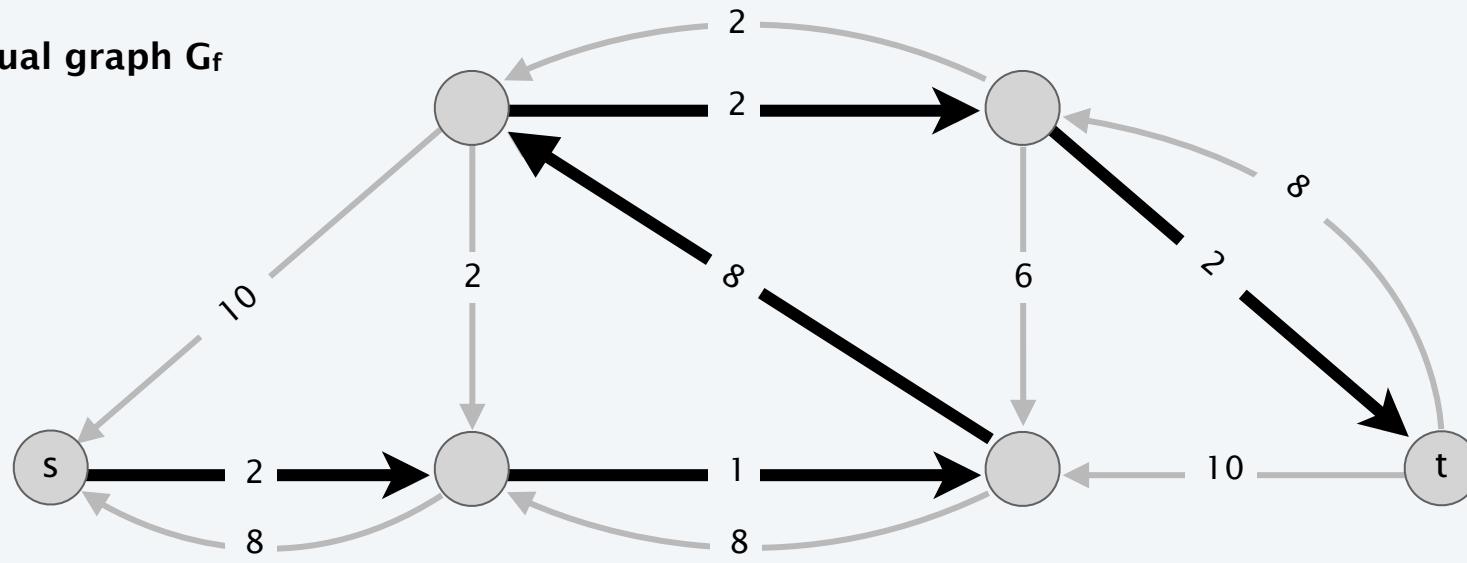


Ford-Fulkerson algorithm demo

network G

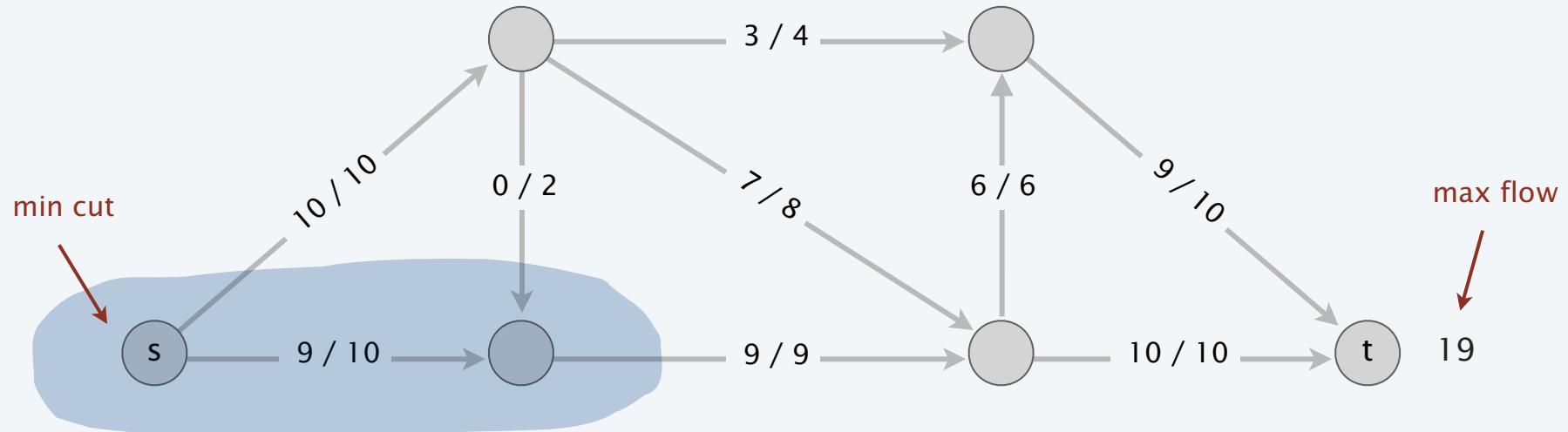


residual graph G_f

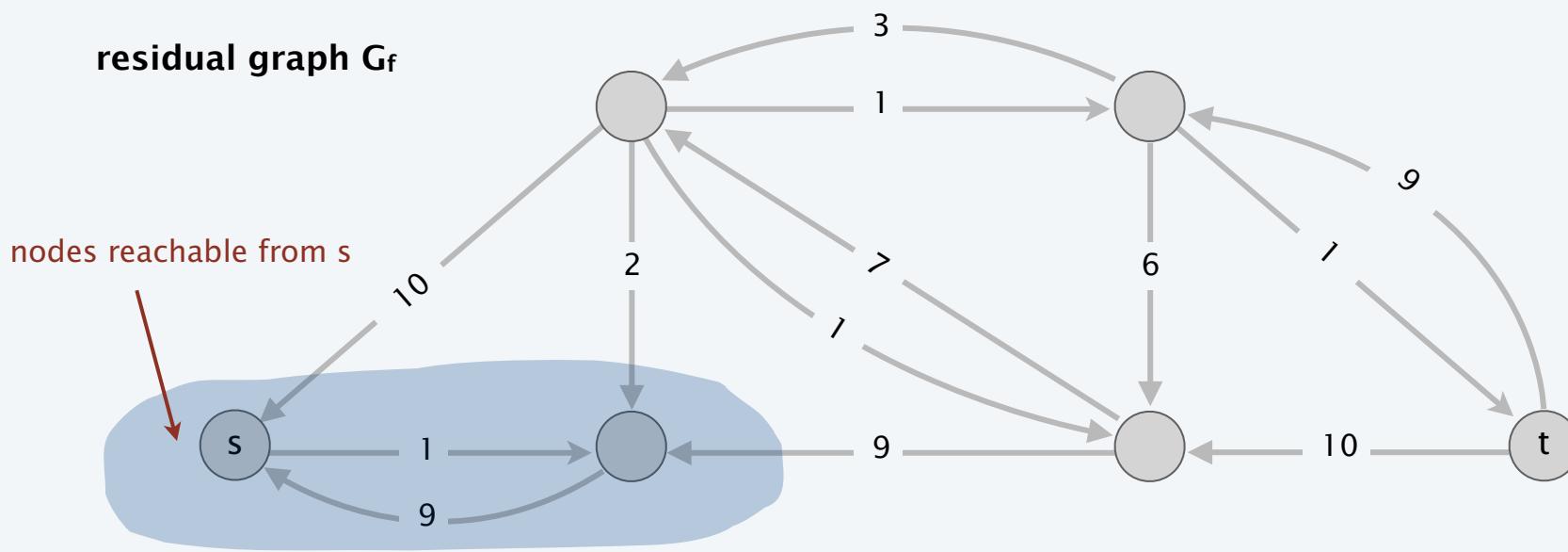


Ford-Fulkerson algorithm demo

network G



residual graph G_f



CSU22012: Data Structures and Algorithms II

Maximum Flow

Ivana.Dusparic@scss.tcd.ie

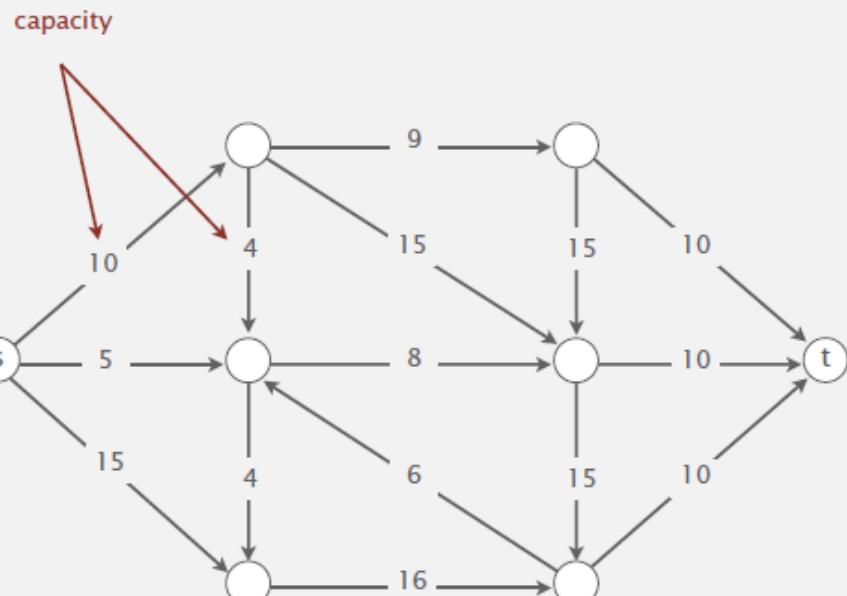
Outline

- › Mincut
- › Maxflow
- › Ford-Fulkerson
- › Edmonds-Karp
- › Applications

Mincut Problem

Input. An edge-weighted digraph, source vertex s , and target vertex t .

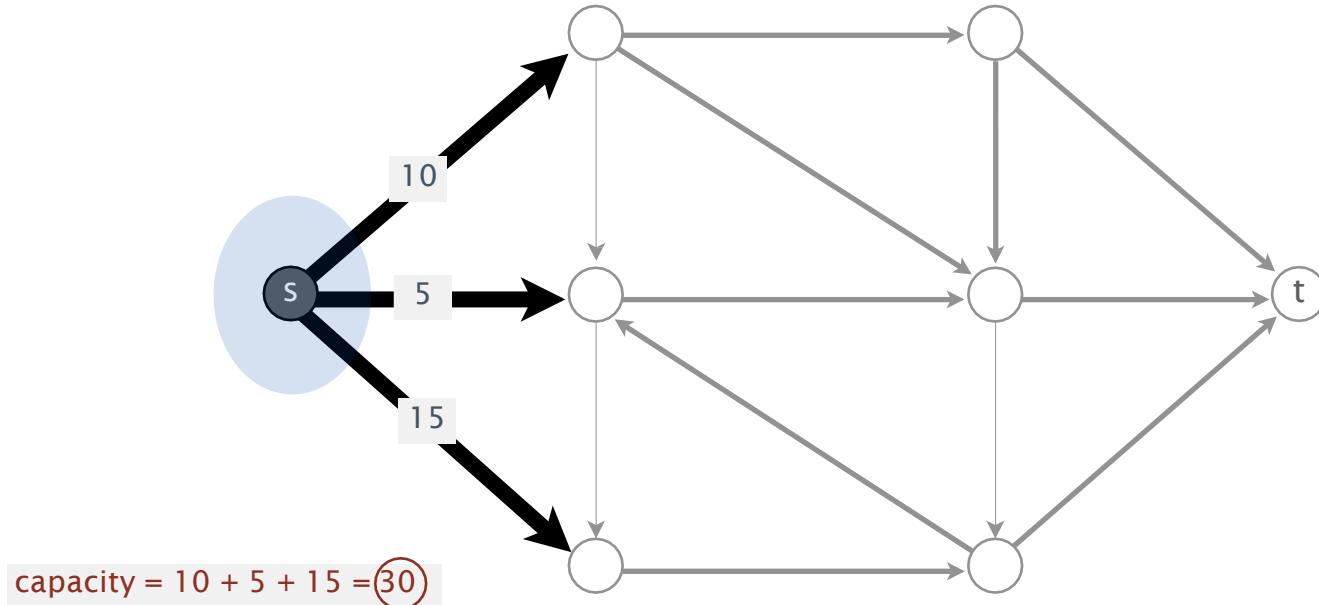
each edge has a
positive capacity



Mincut problem

Def. A *st-cut (cut)* is a partition of the vertices into two disjoint sets, with s in one set A and t in the other set B .

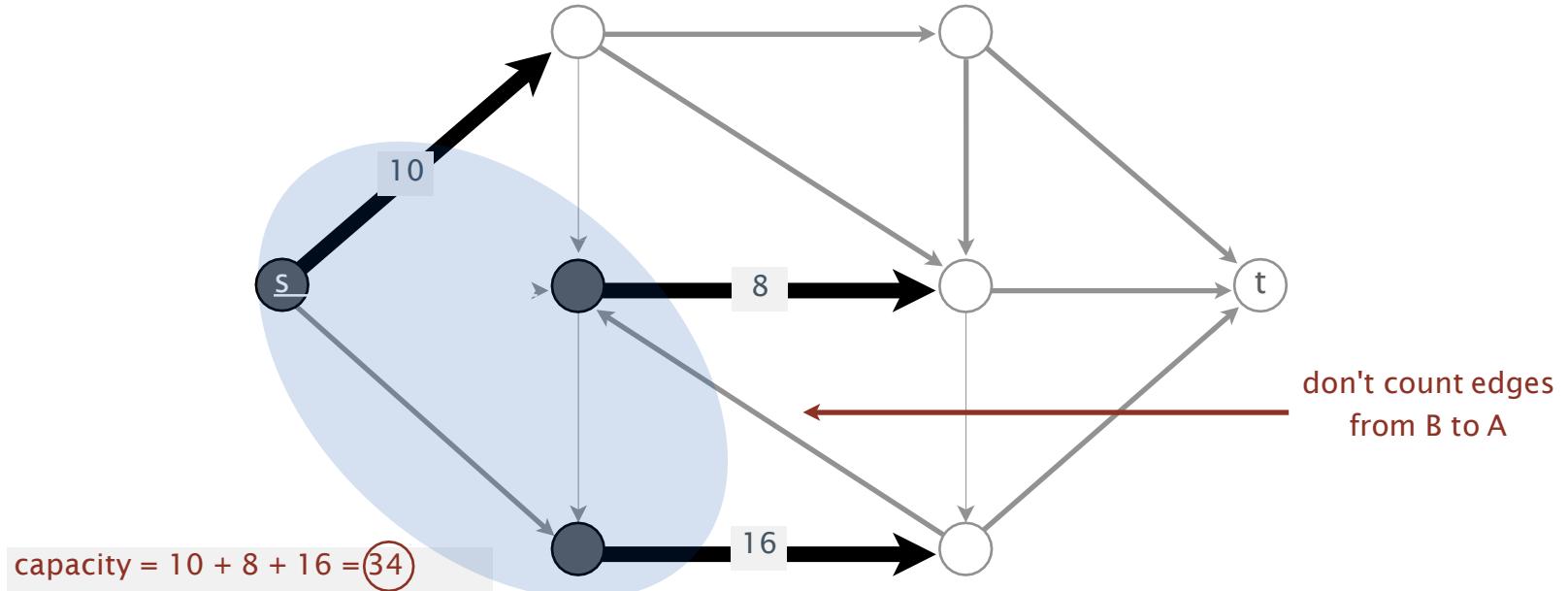
Def. Its **capacity** is the sum of the capacities of the edges from A to B .



Mincut problem

Def. A *st-cut (cut)* is a partition of the vertices into two disjoint sets, with s in one set A and t in the other set B .

Def. Its *capacity* is the sum of the capacities of the edges from A to B .

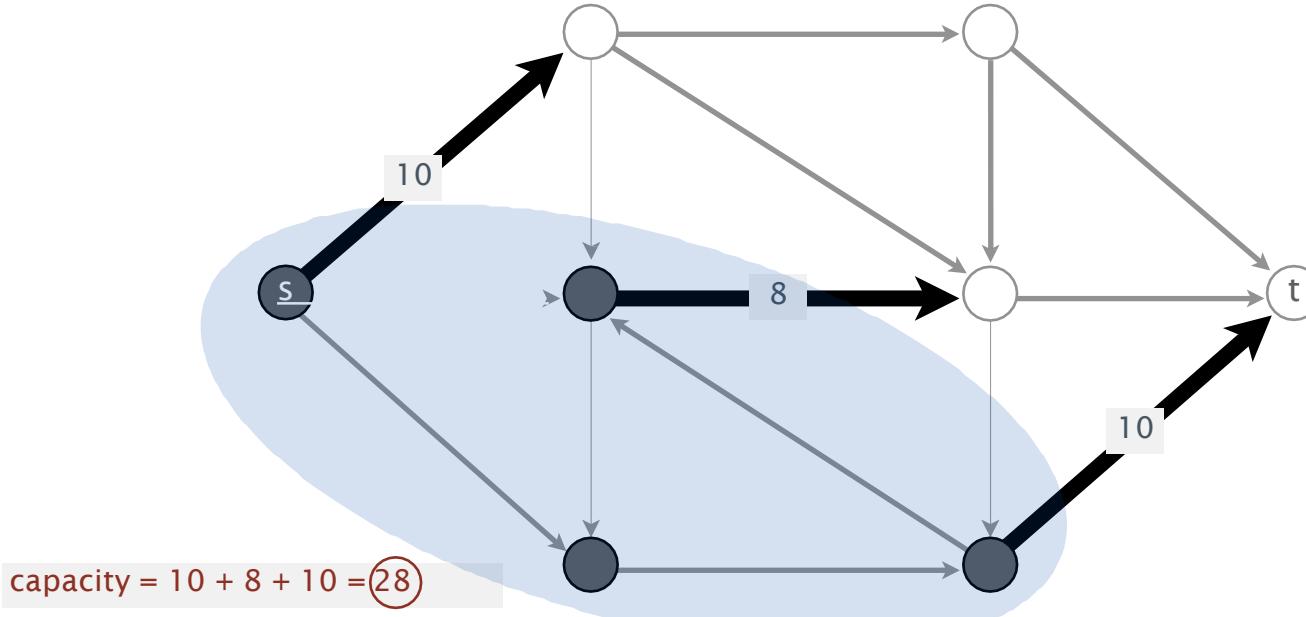


Mincut problem

Def. A *st-cut (cut)* is a partition of the vertices into two disjoint sets, with s in one set A and t in the other set B .

Def. Its **capacity** is the sum of the capacities of the edges from A to B .

Minimum st-cut (mincut) problem. Find a cut of minimum capacity.

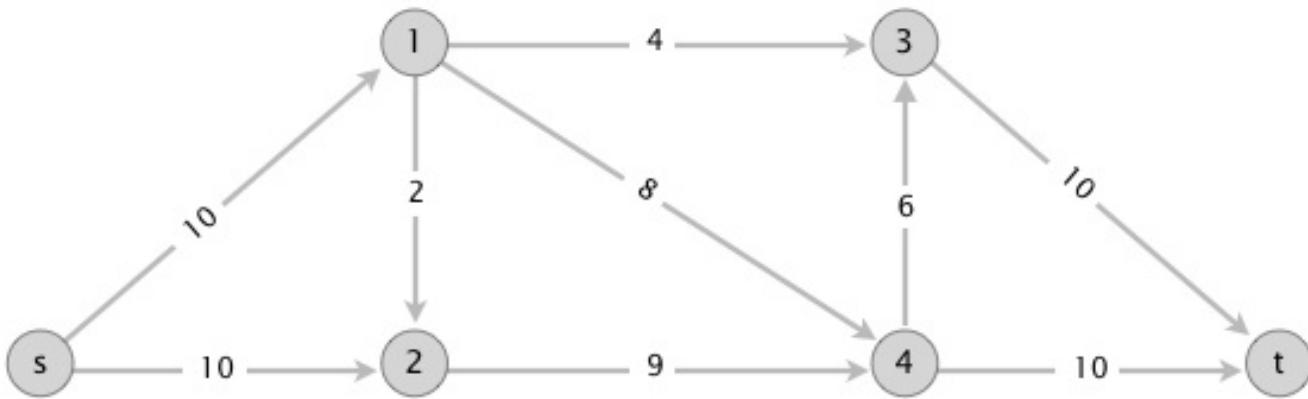


Mincut applications

- › Image segmentation – background and foreground
- › Cutting of road/rail/information network
- › Splitting large-graphs/sharding – if too large to compute
- › Community detection in social media – cut along least interactions, or common interests etc
- › Etc

Mincut exercise

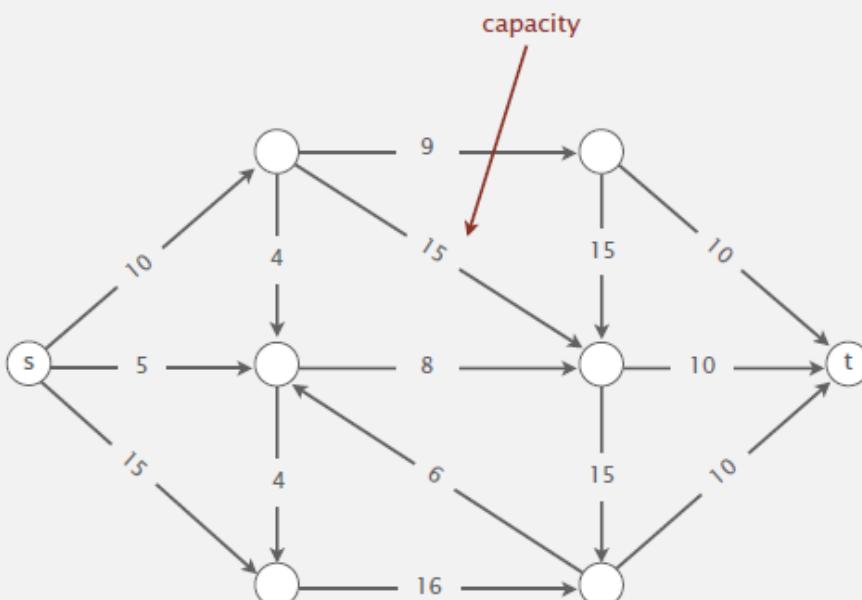
In the flow network below, what is the capacity of the cut with $A = \{s, 2, 4\}$ and $B = \{1, 3, t\}$?



Maxflow problem

Input. An edge-weighted digraph, source vertex s , and target vertex t .

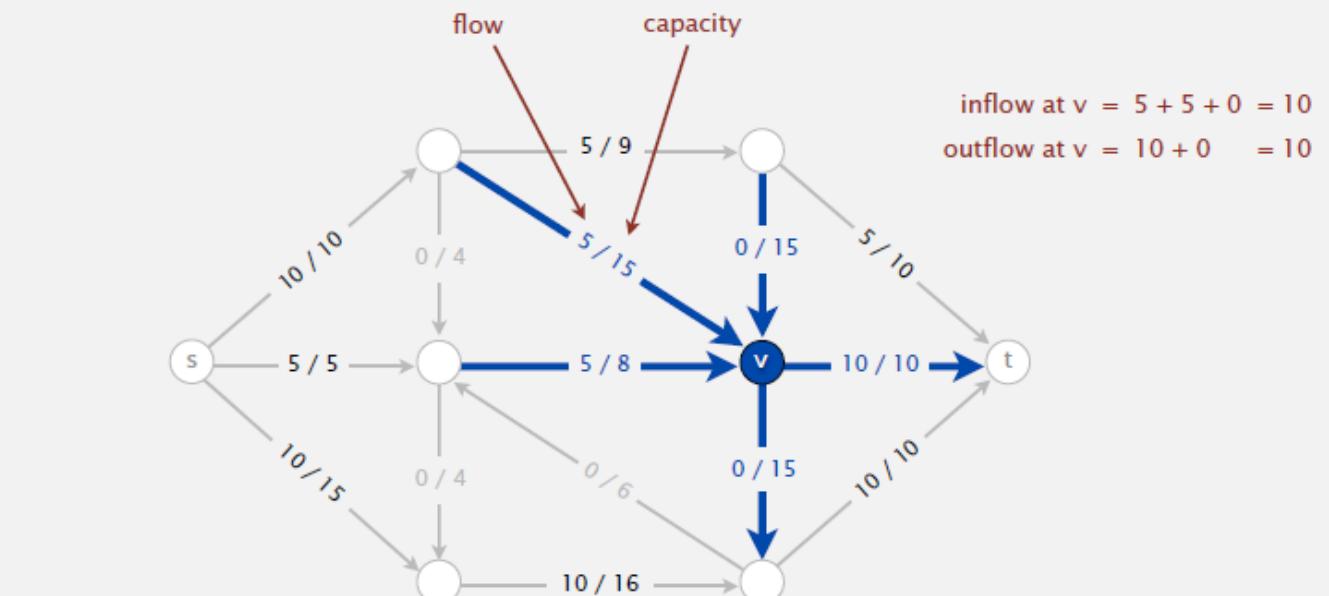
each edge has a
positive capacity



Maxflow

Def. An st -flow (flow) is an assignment of values to the edges such that:

- Capacity constraint: $0 \leq$ edge's flow \leq edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except s and t).



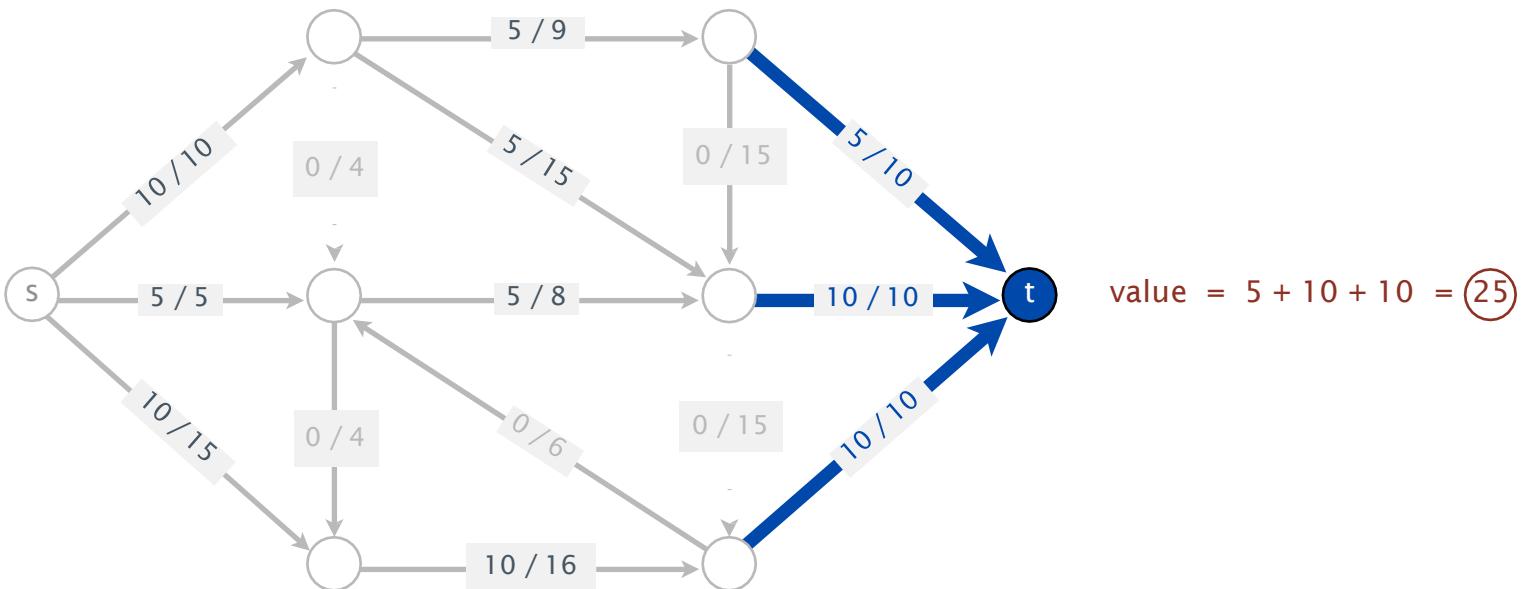
Maxflow problem

Def. An ***st*-flow (flow)** is an assignment of values to the edges such that:

- Capacity constraint: $0 \leq$ edge's flow \leq edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except *s* and *t*).

Def. The **value** of a flow is the inflow at *t*.

we assume no edges point to *s* or from *t*



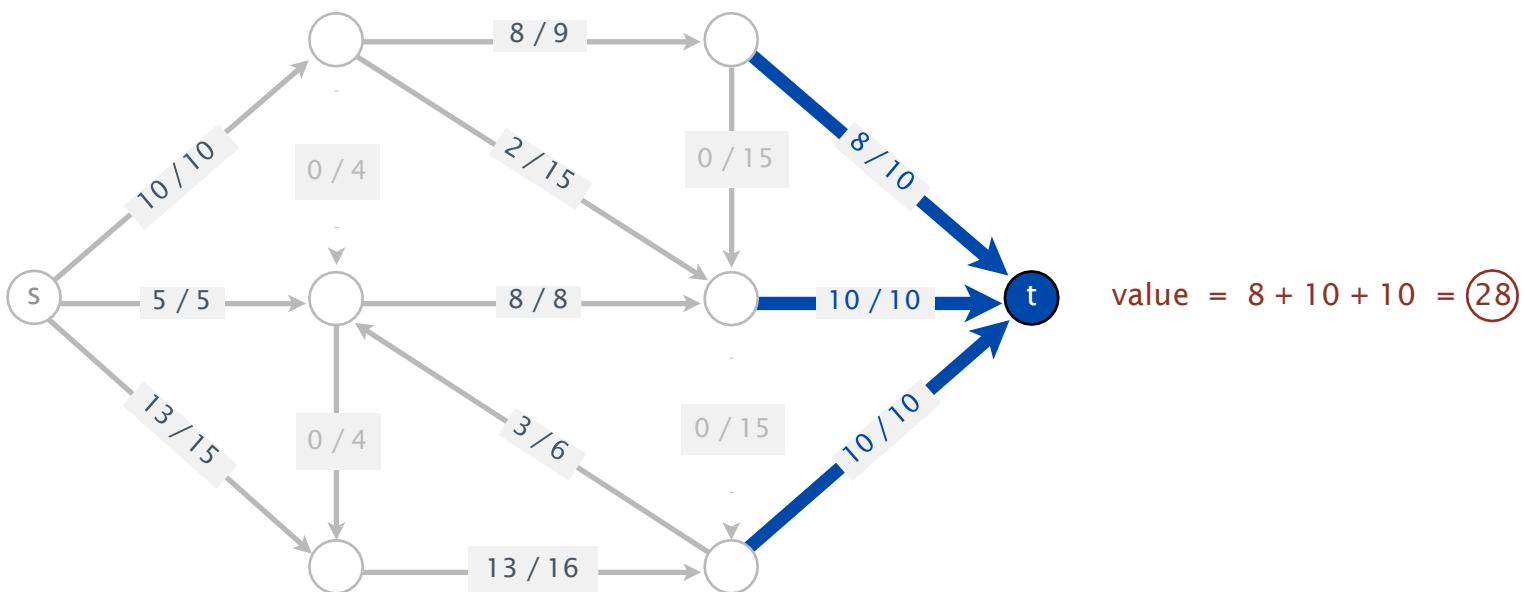
Maxflow problem

Def. An ***st*-flow (flow)** is an assignment of values to the edges such that:

- Capacity constraint: $0 \leq$ edge's flow \leq edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except s and t).

Def. The **value** of a flow is the inflow at t .

Maximum ***st*-flow (maxflow)** problem. Find a flow of maximum value.

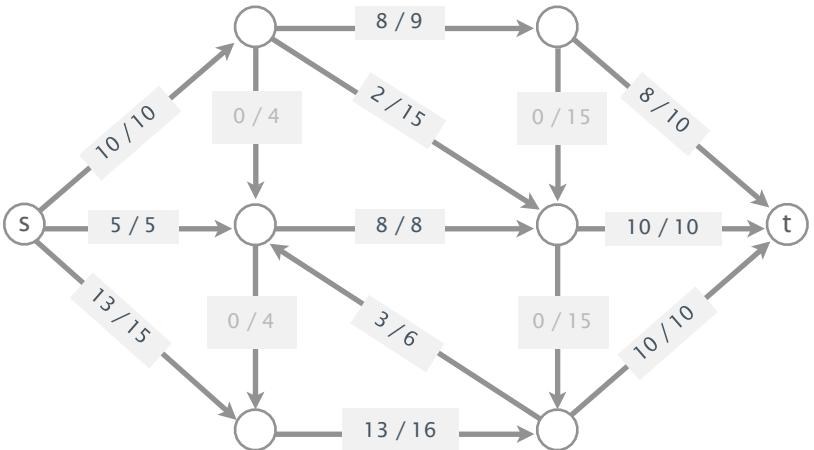


Summary

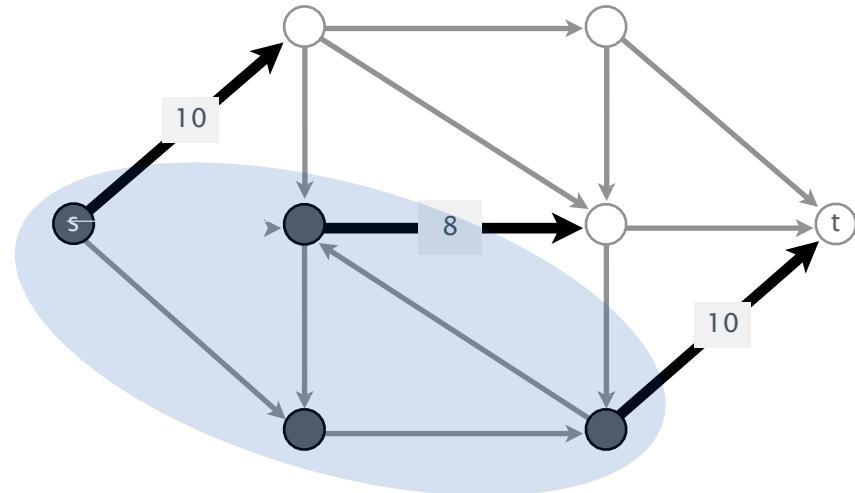
Input. A weighted digraph, source vertex s , and target vertex t .

Mincut problem. Find a cut of minimum capacity.

Maxflow problem. Find a flow of maximum value.



value of flow = 28



capacity of cut = 28

Remarkable fact. These two problems are dual!

Ford-Fulkerson method

Residual graph

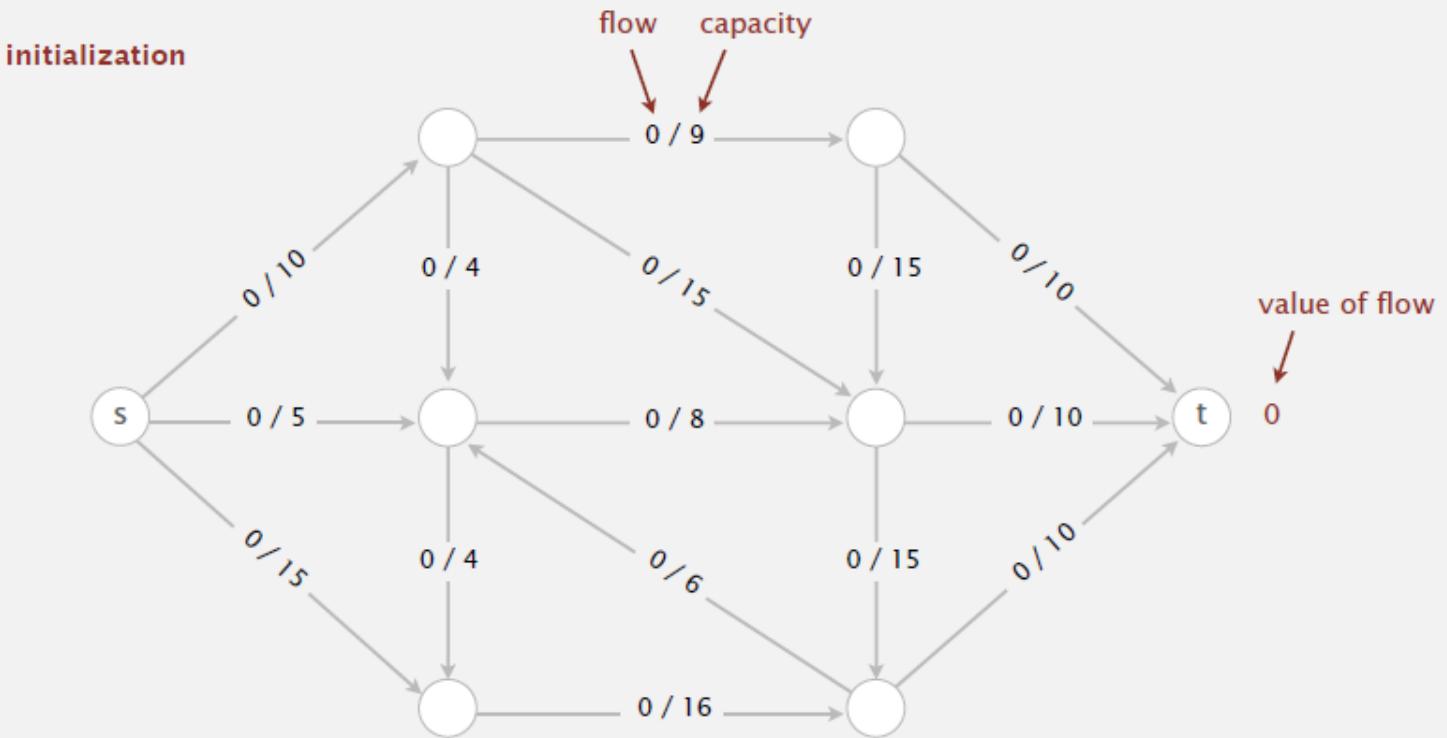
- › Residual capacity of an edge – difference between capacity and flow
- › Residual graph:
 - same vertices as the original network
 - one or two edges for each edge in the original
 - if the flow along the edge $x-y$ is less than the capacity there is a forward edge $x-y$ with a capacity equal to the difference between the capacity and the flow (this is called the residual capacity)
 - if the flow is positive there is a backward edge $y-x$ with a capacity equal to the flow on $x-y$.

Augmenting path

- › a path from the source to the sink in the residual network, whose purpose is to increase the flow in the original one
- › The edges in this path can point the “wrong way” according to the original network.
- › The path capacity of a path is the minimum capacity of an edge along that path.

Ford-Fulkerson

Initialization. Start with 0 flow.

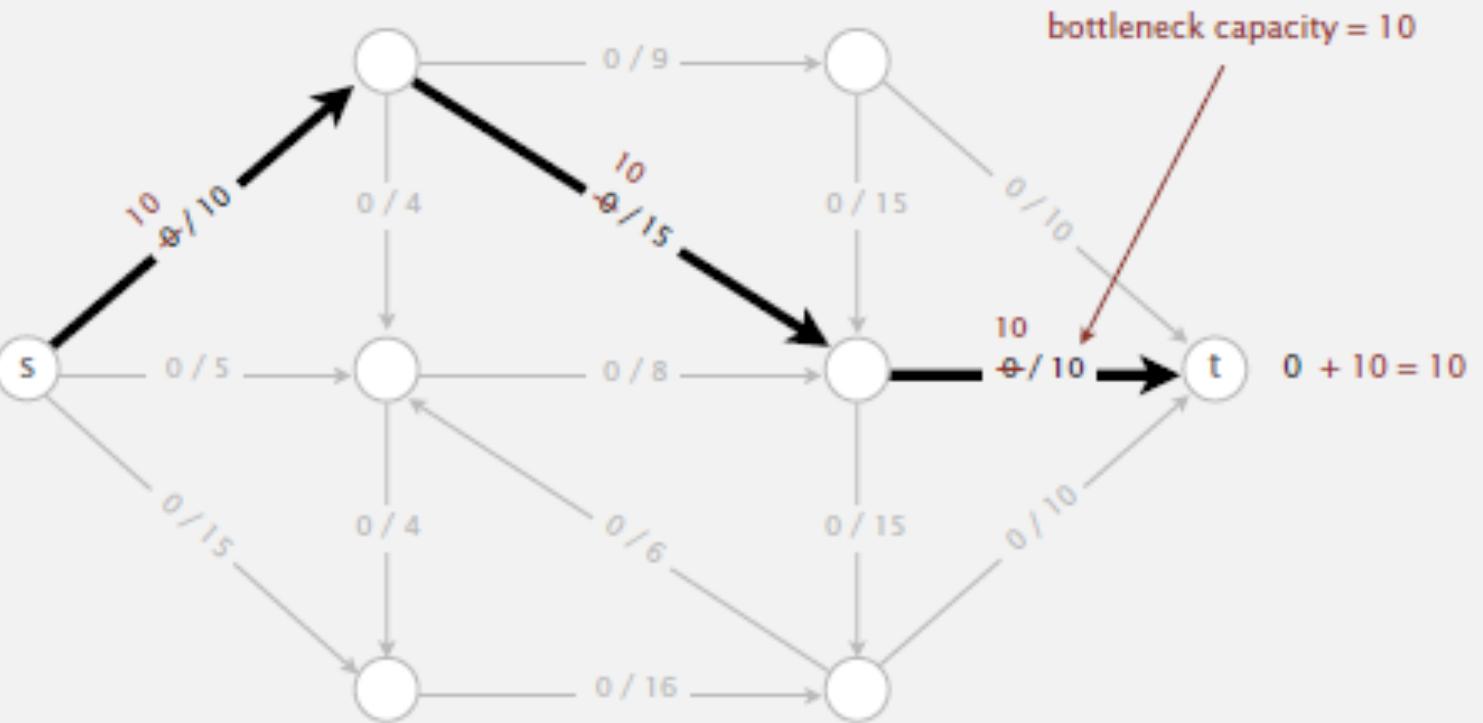


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

1st augmenting path

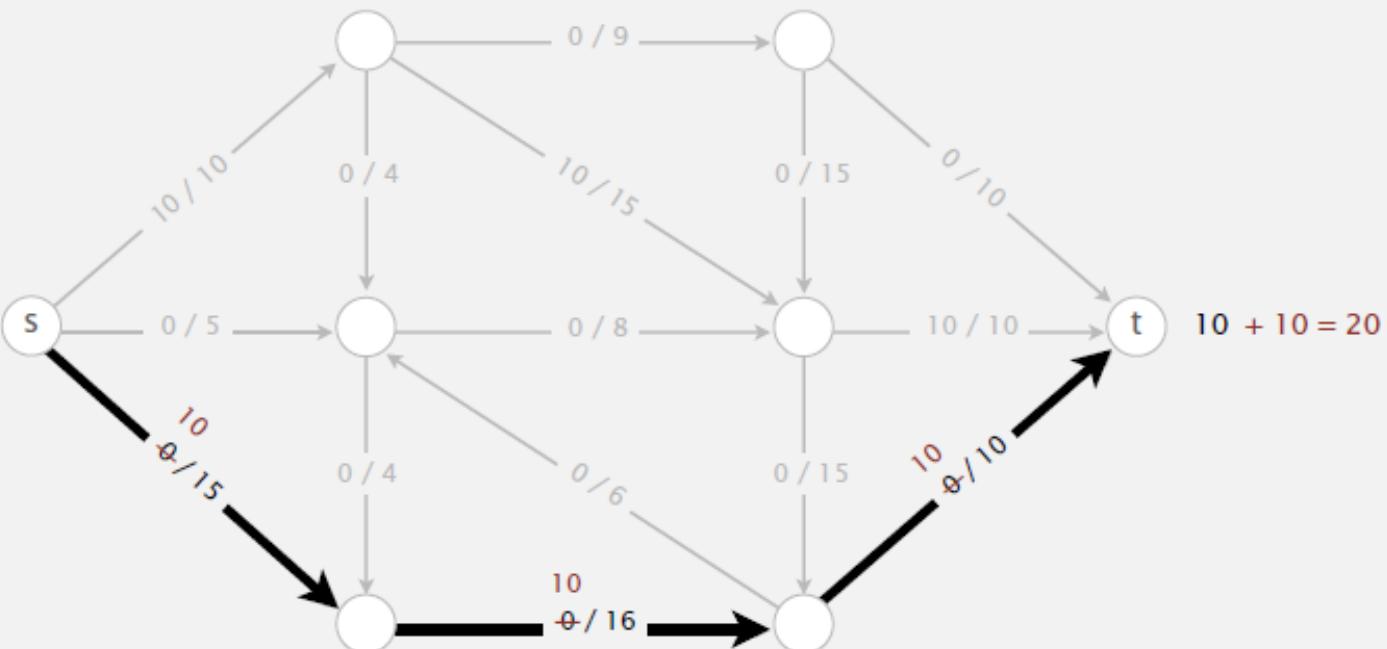


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

2nd augmenting path

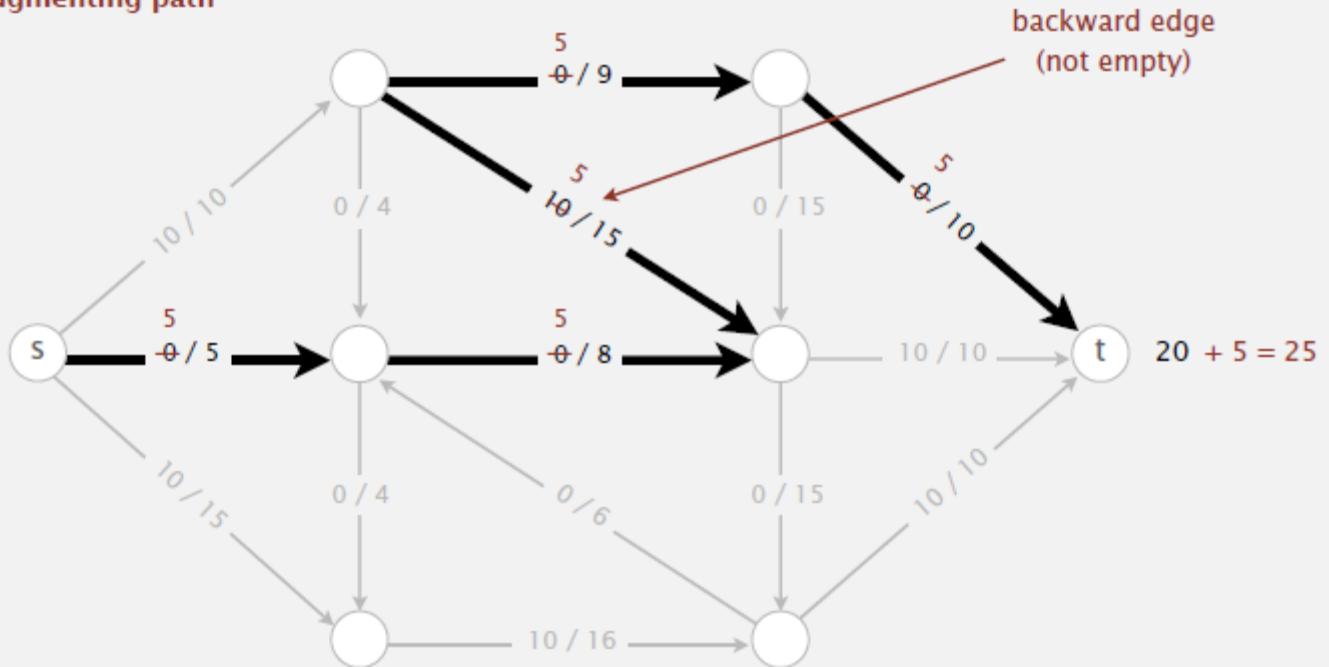


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

3rd augmenting path

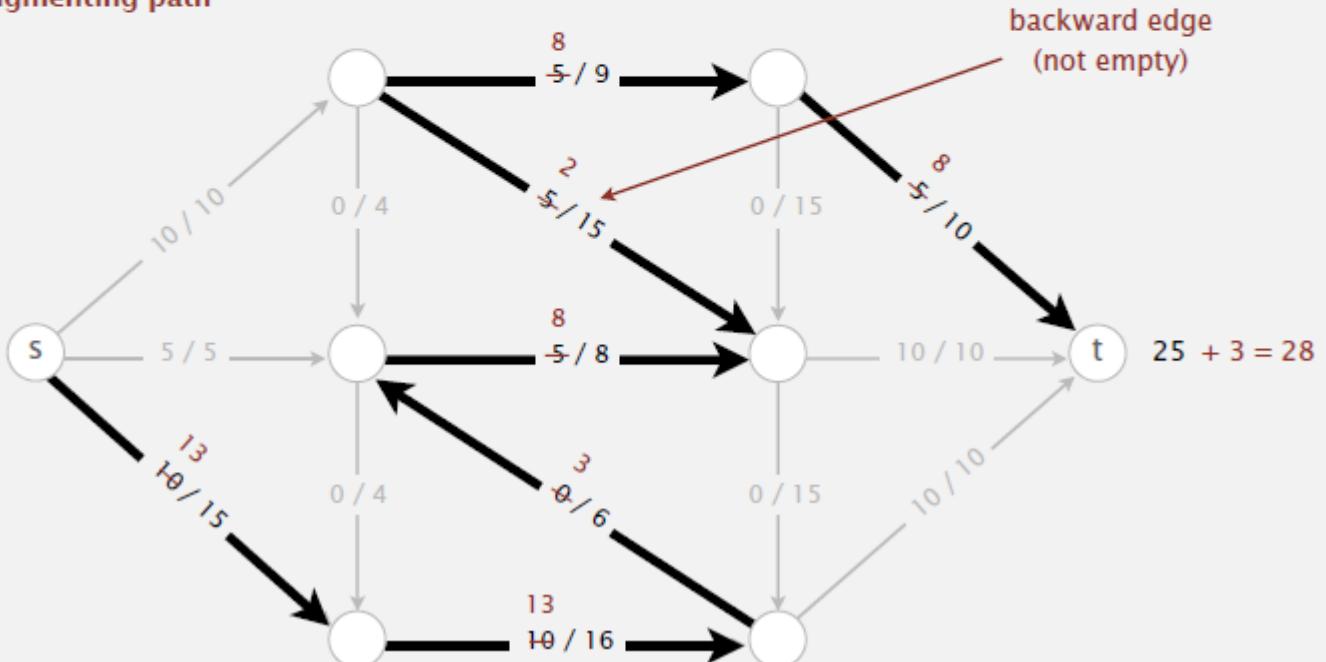


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

4th augmenting path

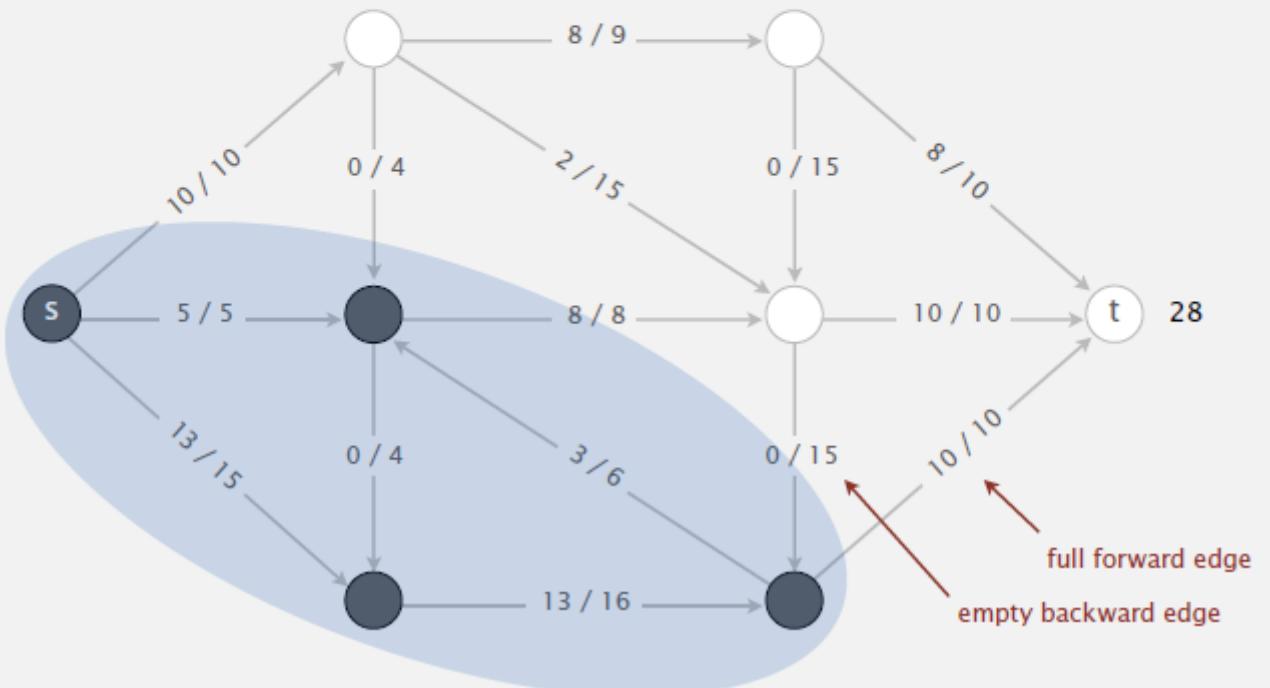


Idea: increase flow along augmenting paths

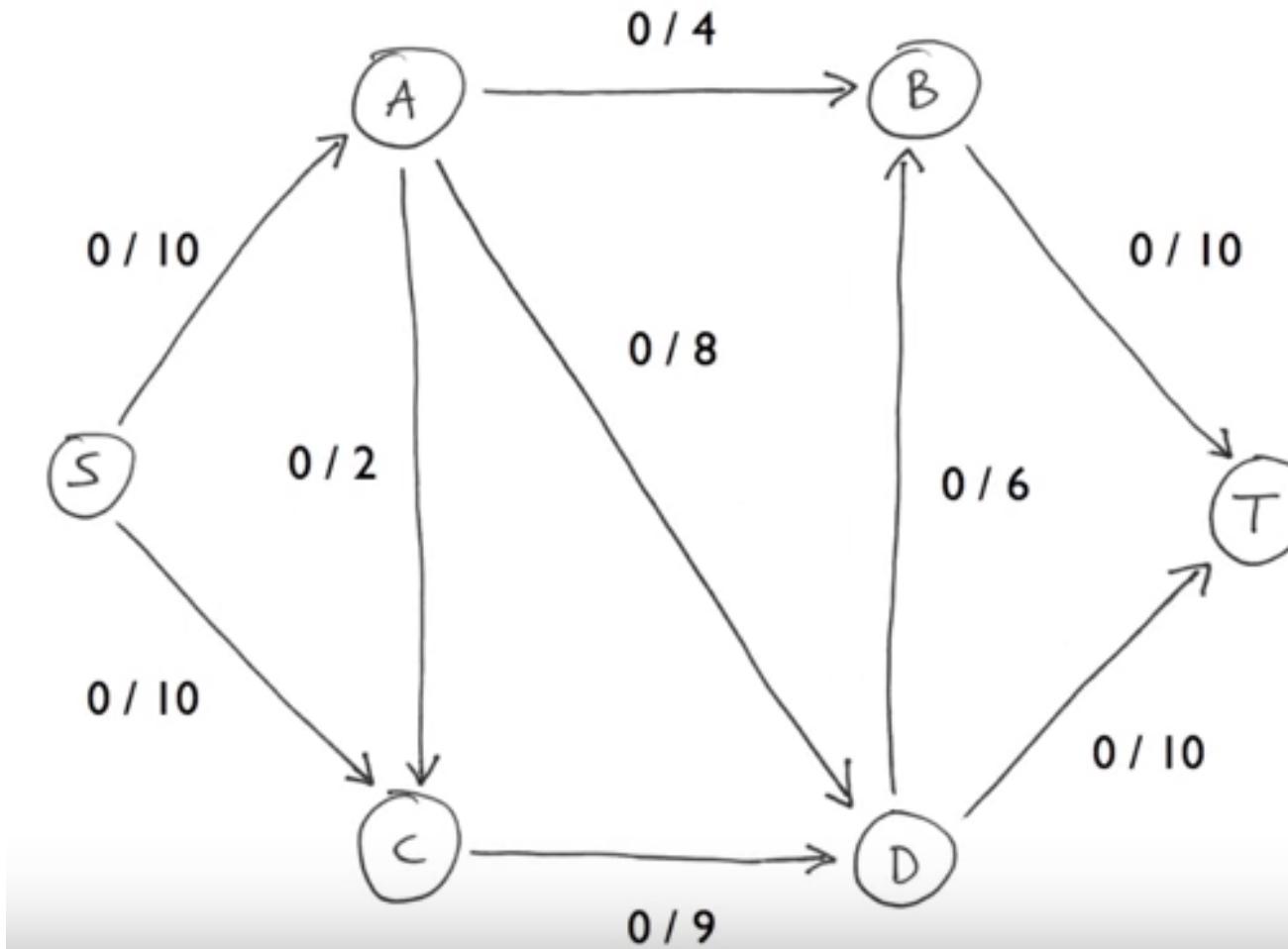
Termination. All paths from s to t are blocked by either a

- Full forward edge.
- Empty backward edge.

no more augmenting paths



Ford Fulkerson exercise



Ford Fulkerson exercise solution

<https://www.youtube.com/watch?v=Tl90tNtKvxs>

Ford-Fulkerson algorithm

Ford-Fulkerson algorithm

Start with 0 flow.

While there exists an augmenting path:

- find an augmenting path
 - compute bottleneck capacity
 - increase flow on that path by bottleneck capacity
-

Questions.

- How to compute a mincut?
- How to find an augmenting path?
- If FF terminates, does it always compute a maxflow?
- Does FF always terminate? If so, after how many augmentations?

How to find mincut from maxflow?

- › mincut – cut of minimum capacity
- › flow – flow into source
- › Maxflow – when no more augmenting paths

Maxflow-mincut theorem

[Augmenting path theorem.](#) A flow f is a maxflow iff no augmenting paths.

[Maxflow-mincut theorem.](#) Value of the maxflow = capacity of mincut.

[Pf.](#) The following three conditions are equivalent for any flow f :

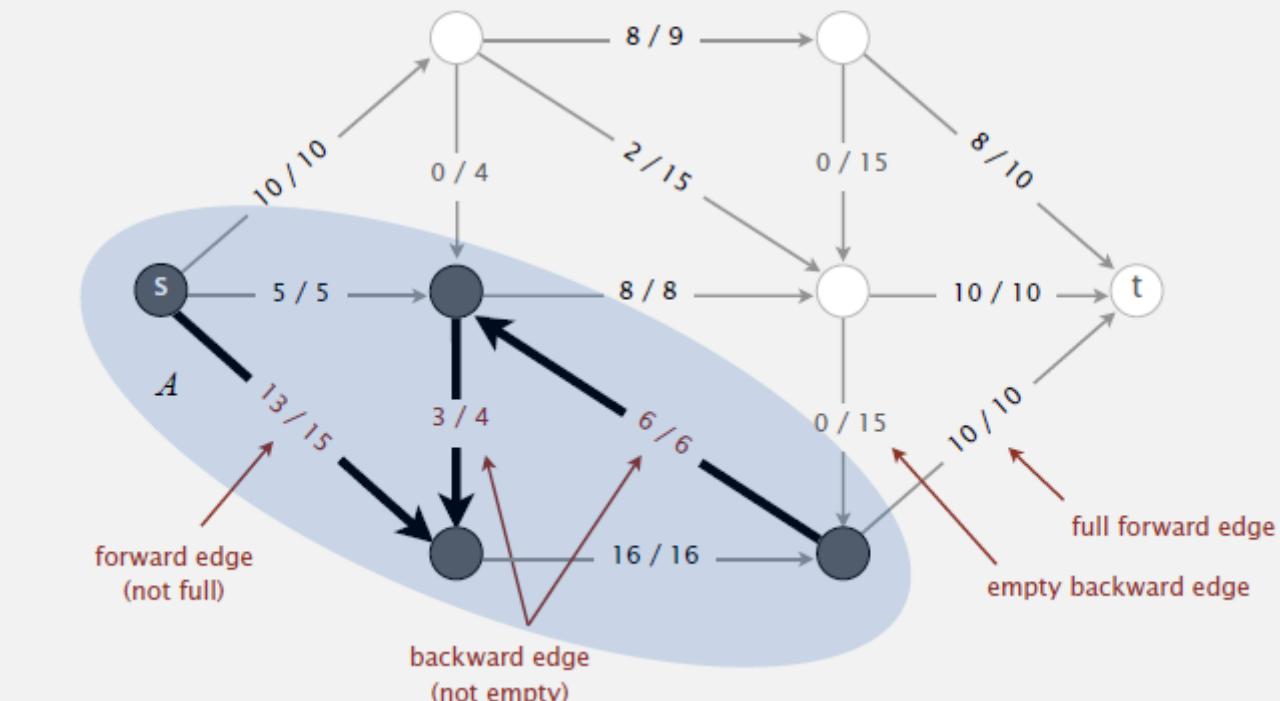
- i. There exists a cut whose capacity equals the value of the flow f .
- ii. f is a maxflow.
- iii. There is no augmenting path with respect to f .

Proof and background in Sedgwick

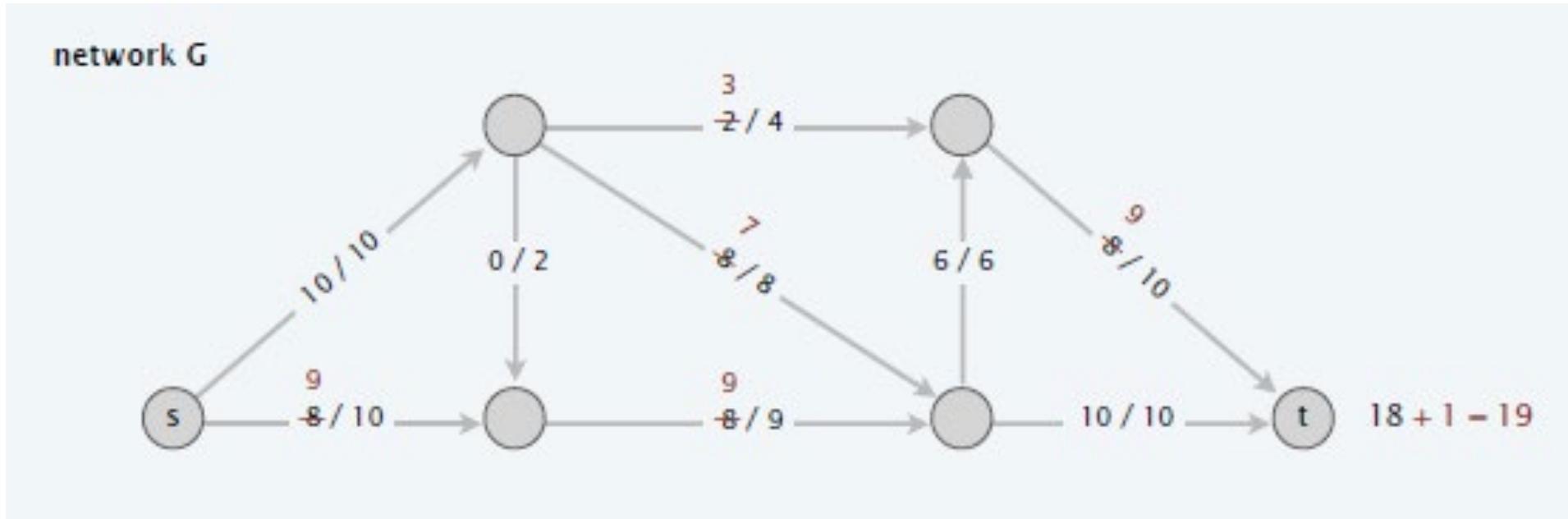
Computing a mincut from a maxflow

To compute mincut (A, B) from maxflow f :

- By augmenting path theorem, no augmenting paths with respect to f .
- Compute $A = \text{set of vertices connected to } s \text{ by an undirected path}$ with no full forward or empty backward edges.



Mincut in exercise



Ford-Fulkerson algorithm

Ford-Fulkerson algorithm

Start with 0 flow.

While there exists an augmenting path:

- find an augmenting path
 - compute bottleneck capacity
 - increase flow on that path by bottleneck capacity
-

Questions.

- How to compute a mincut? Easy.
- How to find an augmenting path? BFS works well.
- If FF terminates, does it always compute a maxflow? Yes.
- Does FF always terminate? If so, after how many augmentations?

yes, provided edge capacities are integers
(or augmenting paths are chosen carefully)

requires clever analysis

Ford-Fulkerson algorithm with integer capacities

Important special case. Edge capacities are integers between 1 and U .

flow on each edge is an integer



Invariant. The flow is **integer-valued** throughout Ford-Fulkerson.

Pf. [by induction]

- Bottleneck capacity is an integer.
- Flow on an edge increases/decreases by bottleneck capacity.

Proposition. Number of augmentations \leq the value of the maxflow.

Pf. Each augmentation increases the value by at least 1.

critical for some applications (stay tuned)

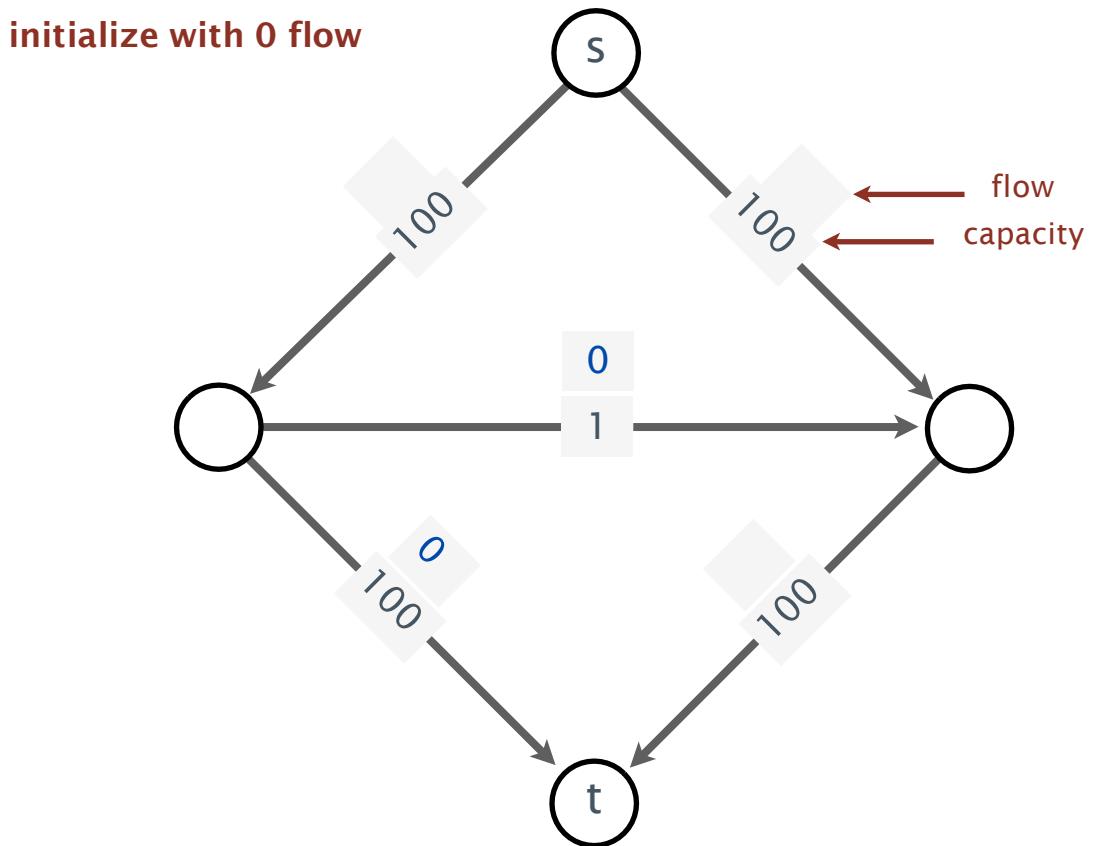


Integrality theorem. There exists an integer-valued maxflow.

Pf. Ford-Fulkerson terminates and maxflow that it finds is integer-valued.

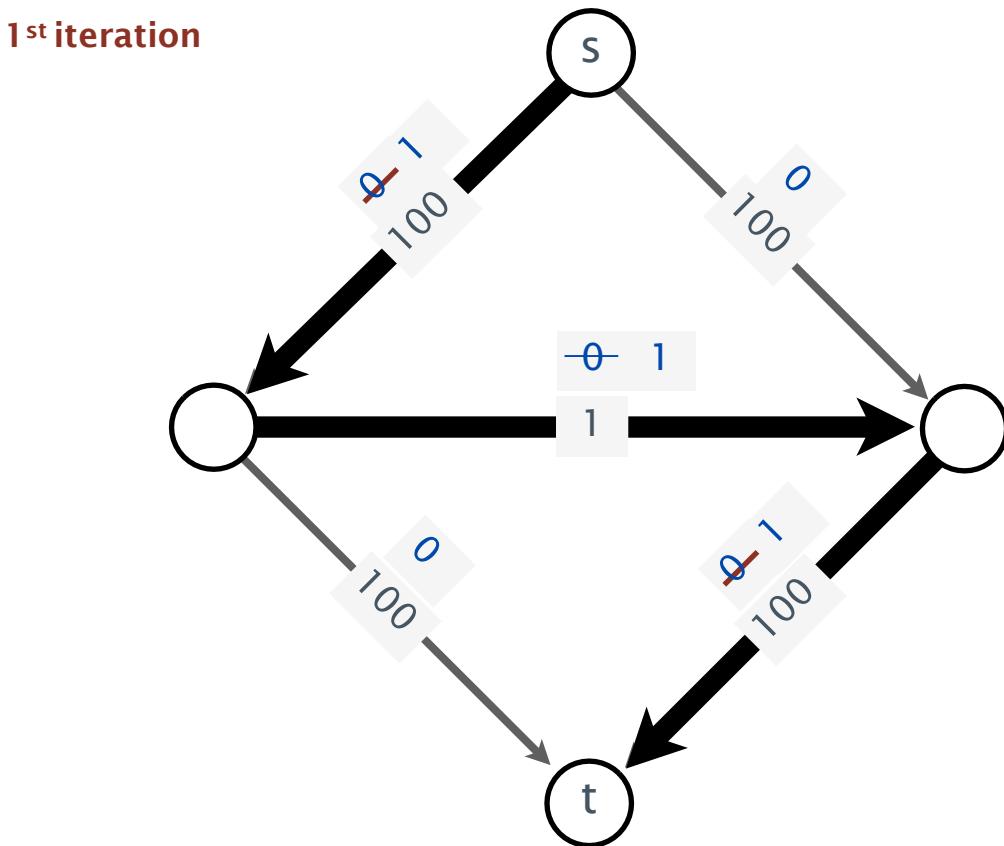
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



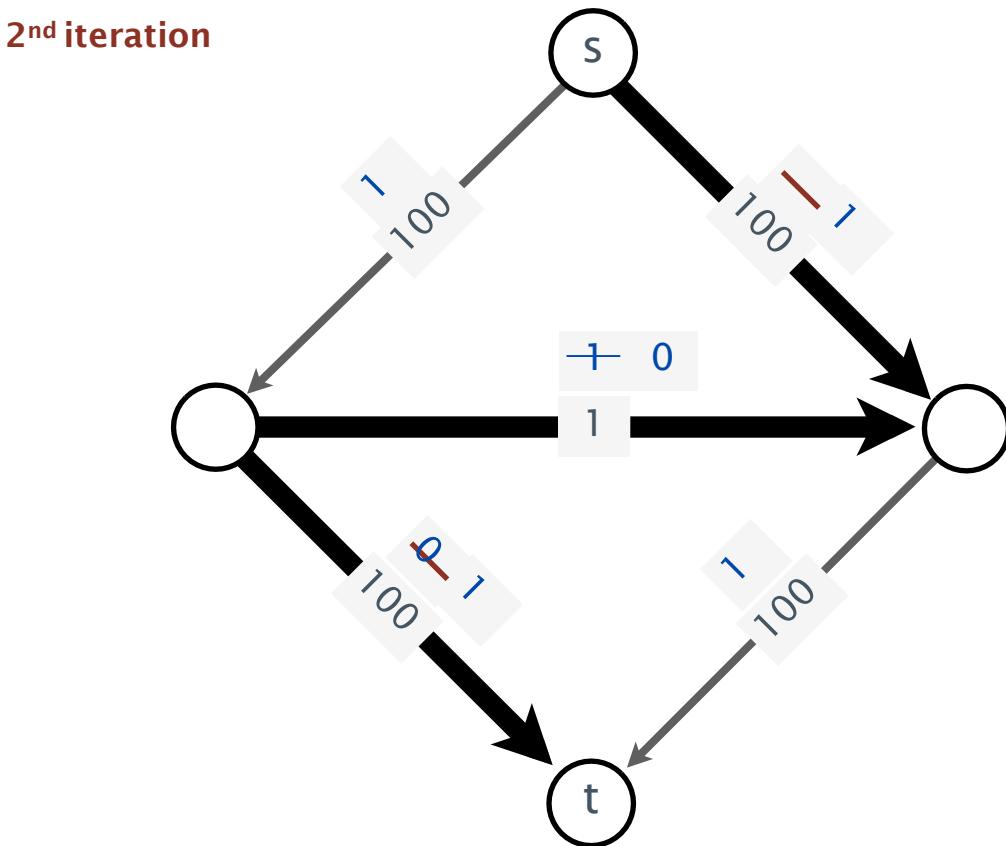
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



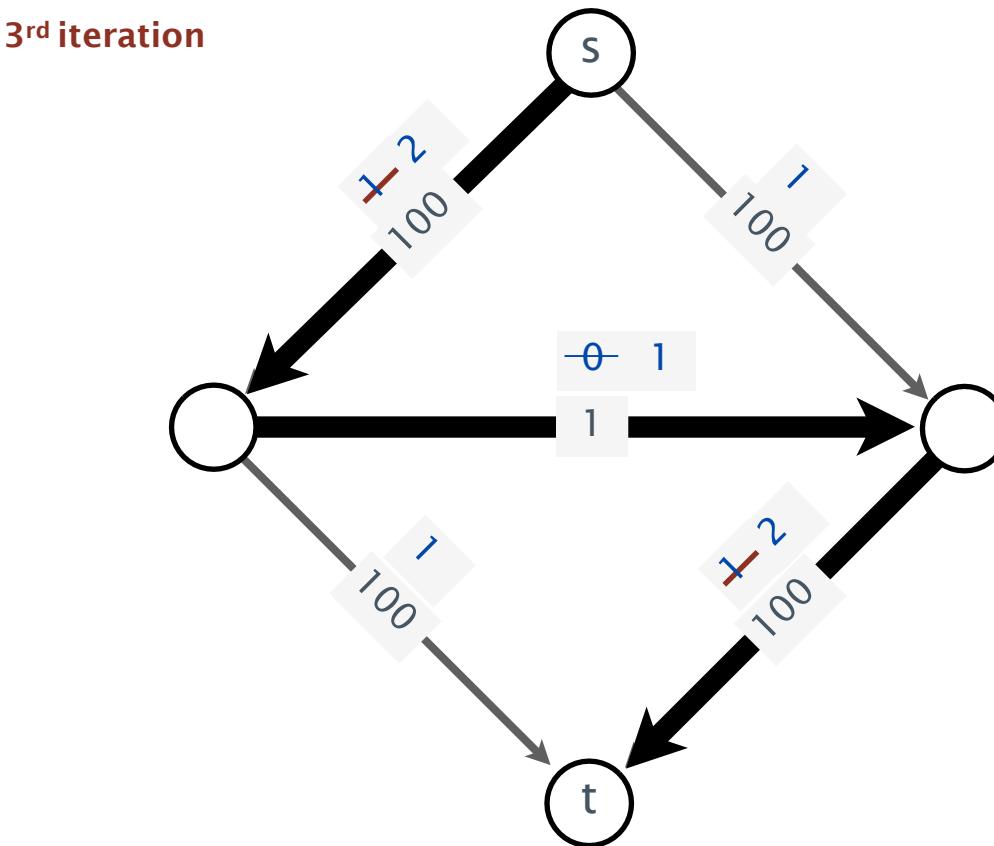
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



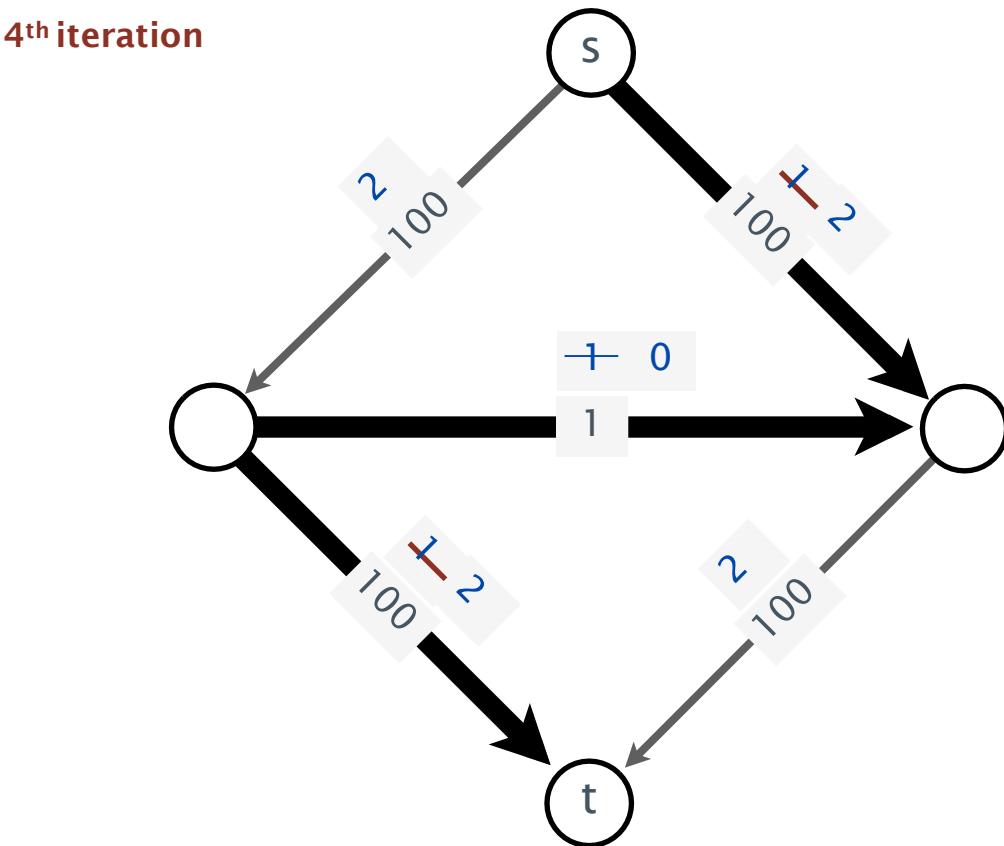
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



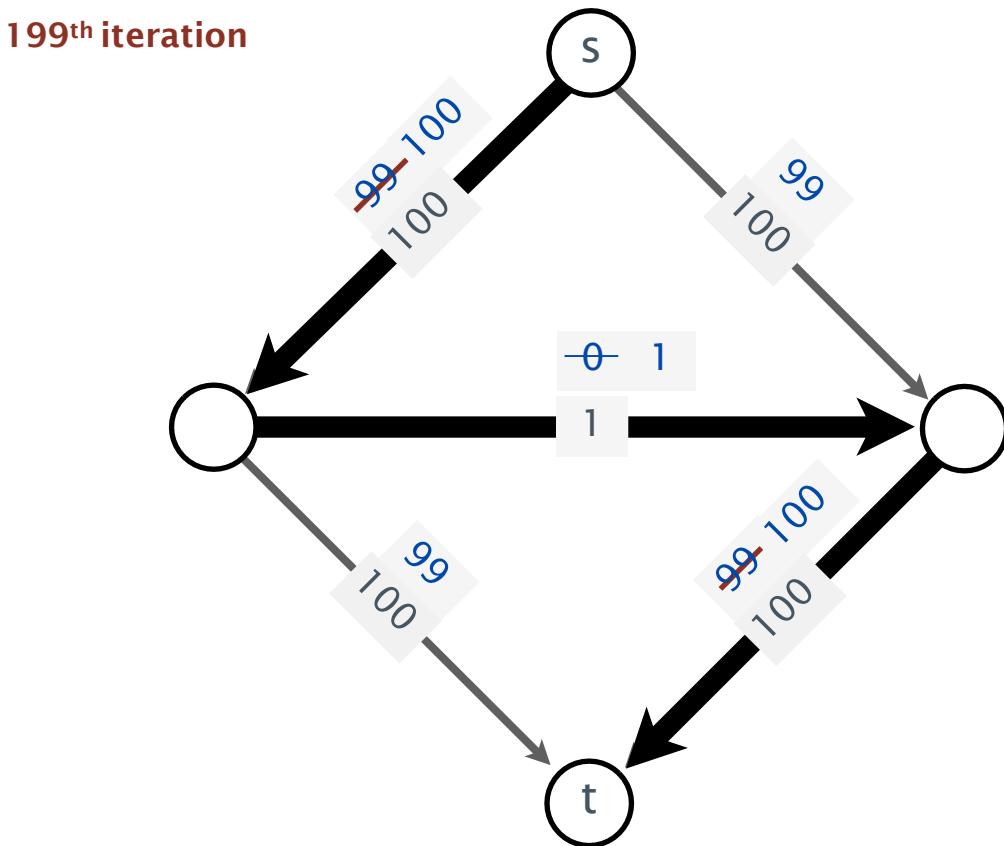
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



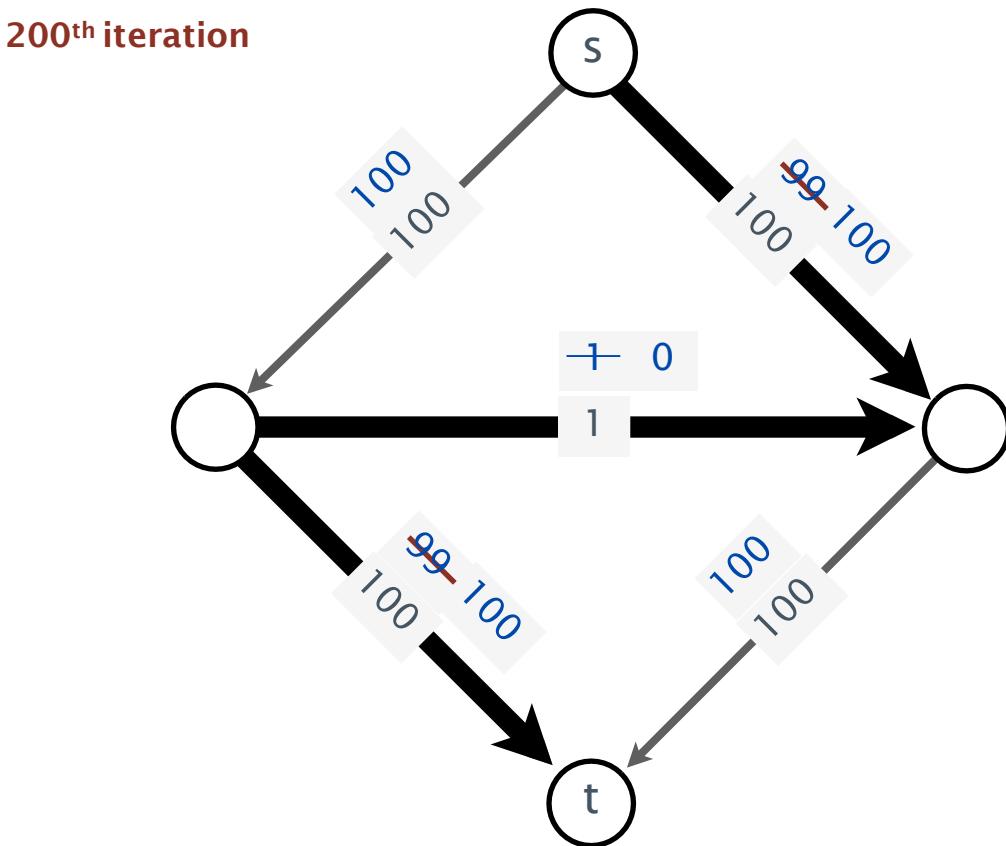
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

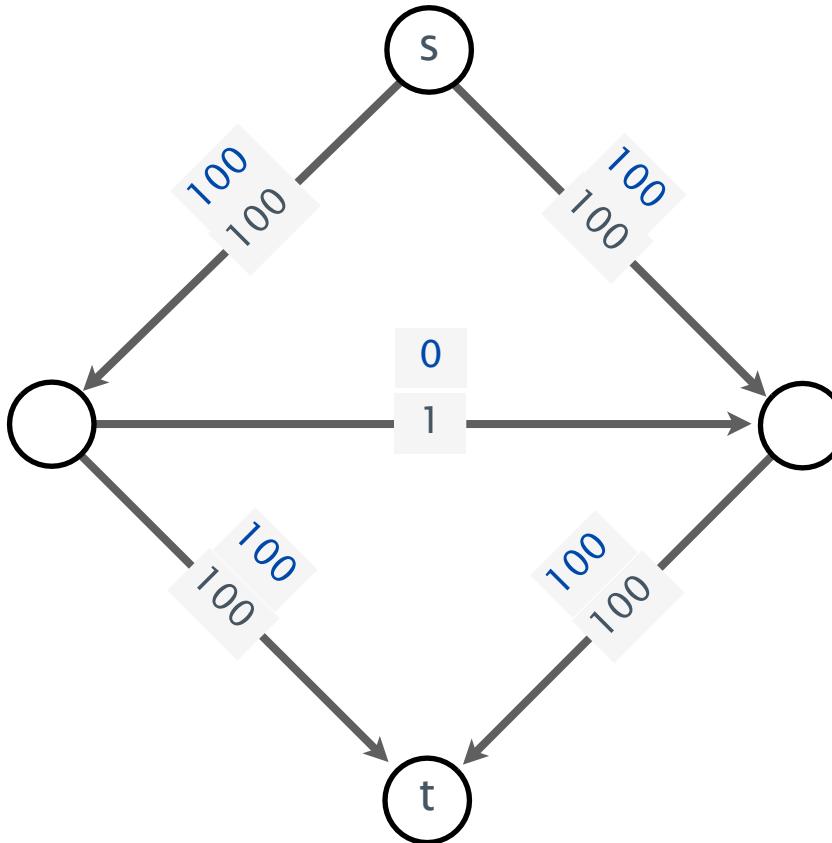


Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

can be exponential in input size

Good news. This case is easily avoided. [use shortest/fattest path]



How to choose augmenting paths?

Use care when selecting augmenting paths.

- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.

augmenting path	number of paths	implementation
random path	$\leq E U$	randomized queue
DFS path	$\leq E U$	stack (DFS)
shortest path	$\leq \frac{1}{2} E V$	queue (BFS)
fattest path	$\leq E \ln(E U)$	priority queue

digraph with V vertices, E edges, and integer capacities between 1 and U

How to choose augmenting paths?

Choose augmenting paths with:

- Shortest path: fewest number of edges.
- Fattest path: max bottleneck capacity.

Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems

JACK EDMONDS

University of Waterloo, Waterloo, Ontario, Canada

AND

RICHARD M. KARP

University of California, Berkeley, California

ABSTRACT. This paper presents new algorithms for the maximum flow problem, the Hitchcock transportation problem, and the general minimum-cost flow problem. Upper bounds on the numbers of steps in these algorithms are derived, and are shown to compare favorably with upper bounds on the numbers of steps required by earlier algorithms.

Edmonds-Karp 1972 (USA)

Dokl. Akad. Nauk SSSR
Tom 194 (1970), No. 4

Soviet Math. Dokl.
Vol. 11 (1970), No. 5

ALGORITHM FOR SOLUTION OF A PROBLEM OF MAXIMUM FLOW IN A NETWORK WITH POWER ESTIMATION

UDC 518.5

E. A. DINIC

Different variants of the formulation of the problem of maximal stationary flow in a network and its many applications are given in [1]. There also is given an algorithm solving the problem in the case where the initial data are integers (or, what is equivalent, commensurable). In the general case this algorithm requires preliminary rounding off of the initial data, i.e. only an approximate solution of the problem is possible. In this connection the rapidity of convergence of the algorithm is inversely proportional to the relative precision.

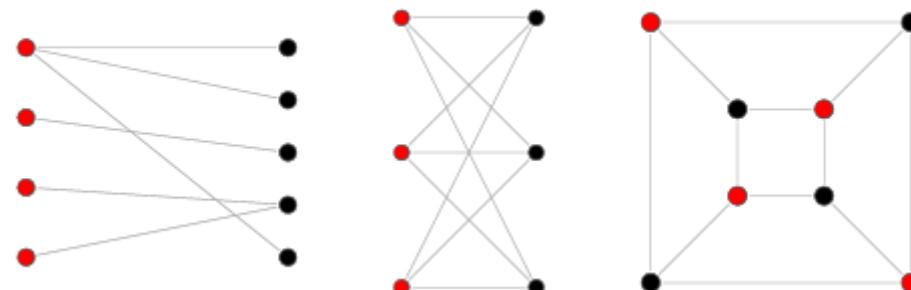
Dinic 1970 (Soviet Union)

Implementation

- › Refer to the book

Applications – bipartite matching

- › Matching applications to jobs, ads to ad slots (with associated revenue), matching jobs to servers in load balancing etc
 - › Bipartite graph – A bipartite graph (bigraph,) is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent
-



Bipartite matching problem

N students apply for N jobs.



Each gets several offers.



Is there a way to match all students to jobs?

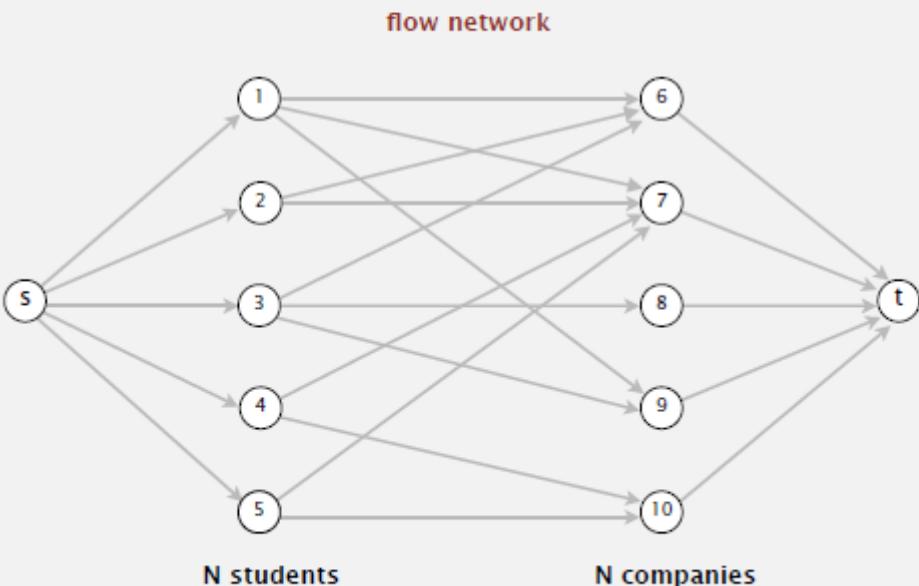


bipartite matching problem

1	Alice	6	Adobe
	Adobe		Alice
	Amazon		Bob
	Google		Carol
2	Bob	7	Amazon
	Adobe		Alice
	Amazon		Bob
3	Carol		Dave
	Adobe		Eliza
	Facebook	8	Facebook
	Google		Carol
4	Dave	9	Google
	Amazon		Alice
	Yahoo		Carol
5	Eliza	10	Yahoo
	Amazon		Dave
	Yahoo		Eliza

Network flow formulation of bipartite matching

- Create s, t , one vertex for each student, and one vertex for each job.
 - Add edge from s to each student (capacity 1).
 - Add edge from each job to t (capacity 1).
 - Add edge from student to each job offered (infinite capacity).



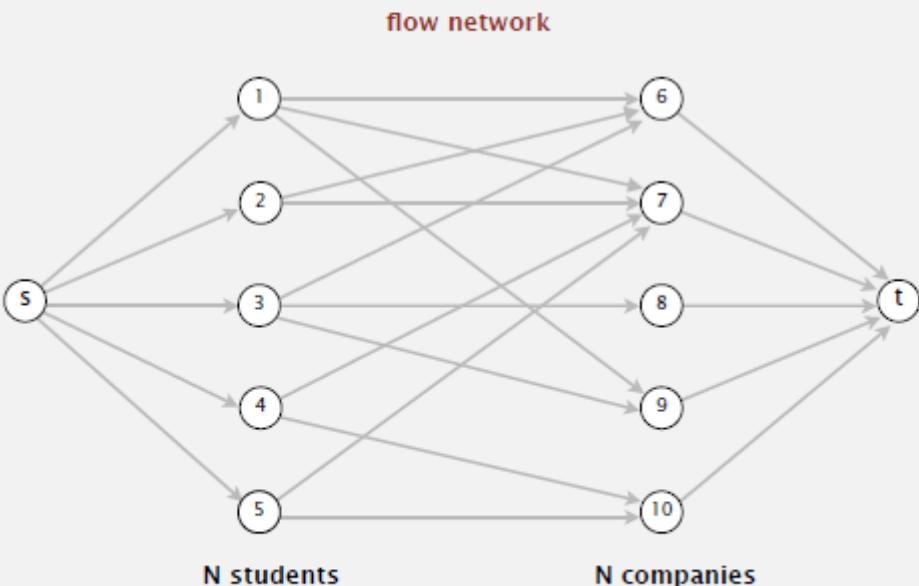
bipartite matching problem			
1	Alice	6	Adobe
	Adobe		Alice
	Amazon		Bob
	Google		Carol
2	Bob	7	Amazon
	Adobe		Alice
	Amazon		Bob
3	Carol		Dave
	Adobe		Eliza
	Facebook	8	Facebook
	Google		Carol
4	Dave	9	Google
	Amazon		Alice
	Yahoo		Carol
5	Eliza	10	Yahoo
	Amazon		Dave
	Yahoo		Eliza

Really good demo of bipartite matching

<http://rosulek.github.io/vamonos/demos/bipartite-matching.html>

Network flow formulation of bipartite matching

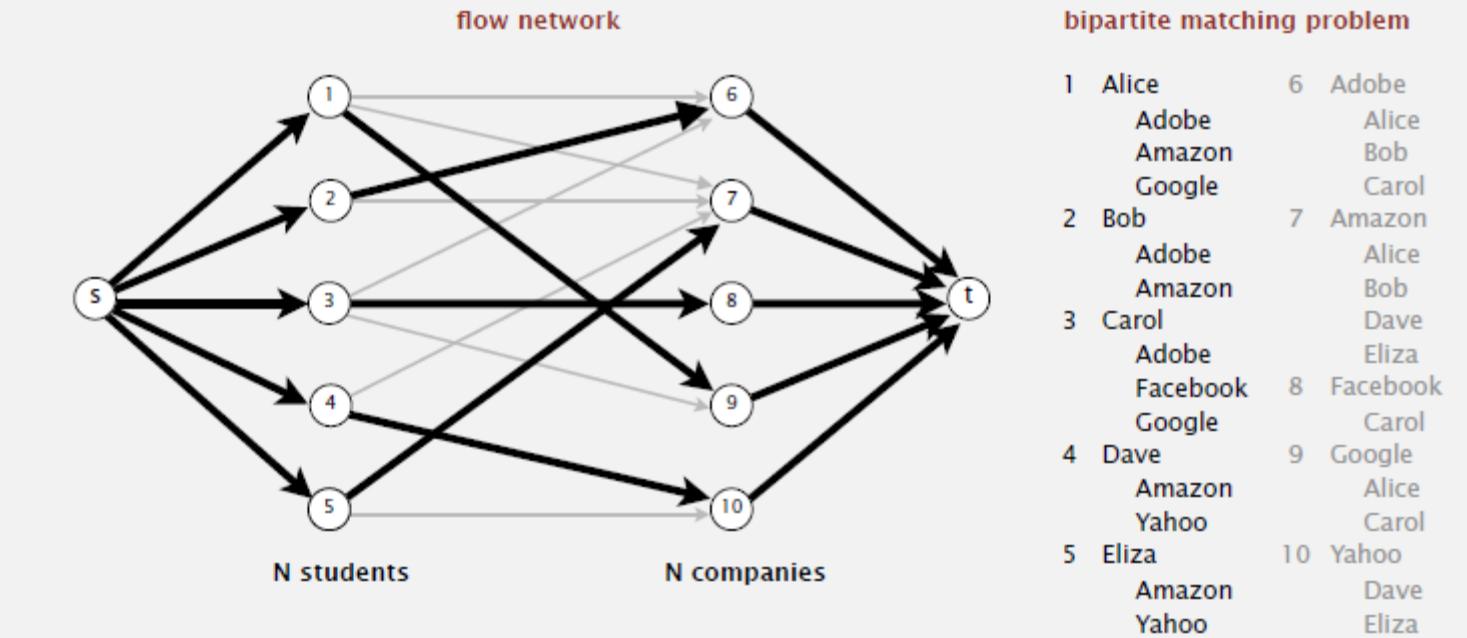
- Create s, t , one vertex for each student, and one vertex for each job.
 - Add edge from s to each student (capacity 1).
 - Add edge from each job to t (capacity 1).
 - Add edge from student to each job offered (infinite capacity).



bipartite matching problem			
1	Alice	6	Adobe
	Adobe		Alice
	Amazon		Bob
	Google		Carol
2	Bob	7	Amazon
	Adobe		Alice
	Amazon		Bob
3	Carol		Dave
	Adobe		Eliza
	Facebook	8	Facebook
	Google		Carol
4	Dave	9	Google
	Amazon		Alice
	Yahoo		Carol
5	Eliza	10	Yahoo
	Amazon		Dave
	Yahoo		Eliza

Network flow formulation of bipartite matching

1-1 correspondence between perfect matchings in bipartite graph and integer-valued maxflows of value N .



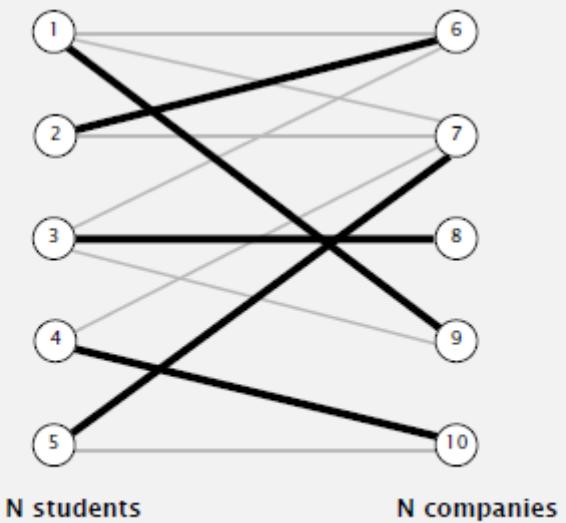
Bipartite matching problem

Given a bipartite graph, find a perfect matching.

perfect matching (solution)

Alice —— Google
Bob —— Adobe
Carol —— Facebook
Dave —— Yahoo
Eliza —— Amazon

bipartite graph

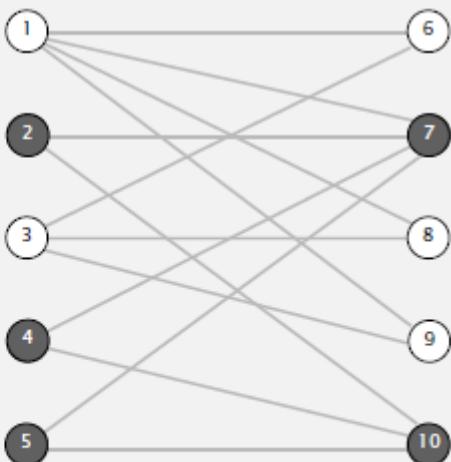


bipartite matching problem

1	Alice	6	Adobe
	Adobe		Alice
	Amazon		Bob
	Google		Carol
2	Bob	7	Amazon
	Adobe		Alice
	Amazon		Bob
3	Carol		Dave
	Adobe		Eliza
	Facebook	8	Facebook
	Google		Carol
4	Dave	9	Google
	Amazon		Alice
	Yahoo		Carol
5	Eliza	10	Yahoo
	Amazon		Dave
	Yahoo		Eliza

What the mincut tells us

Goal. When no perfect matching, explain why.



$$S = \{ 2, 4, 5 \}$$

$$T = \{ 7, 10 \}$$

student in S
can be matched
only to
companies in T

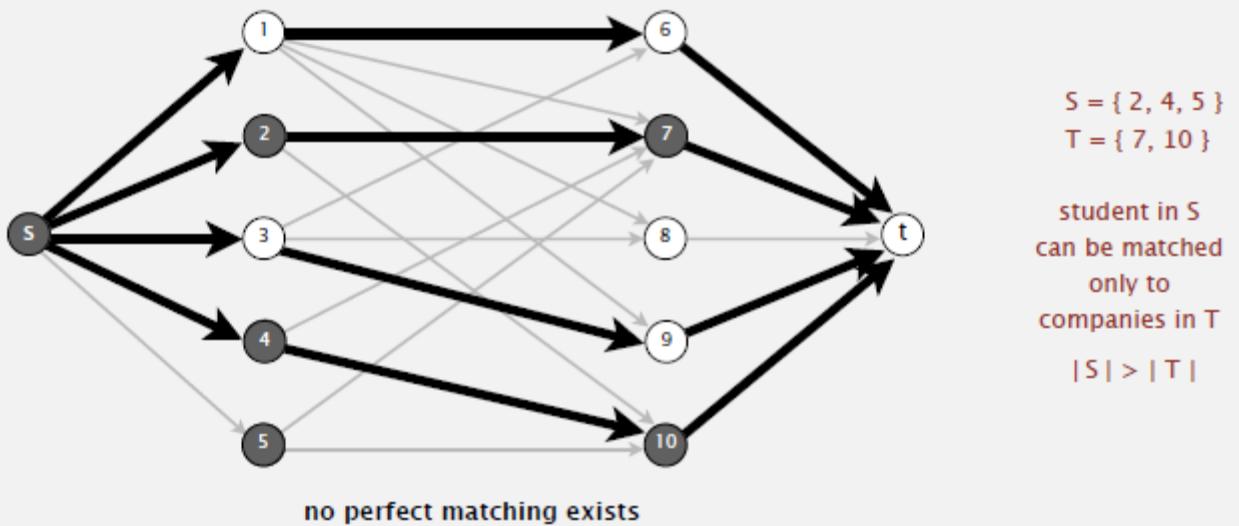
$$|S| > |T|$$

no perfect matching exists

What the mincut tells us

Mincut. Consider mincut (A, B) .

- Let S = students on s side of cut.
- Let T = companies on s side of cut.
- Fact: $|S| > |T|$; students in S can be matched only to companies in T .



Bottom line. When no perfect matching, mincut explains why.

Multi-source multi-sink networks?

- › Add super source and super sink

Additional material

- › Good tutorial

<https://www.topcoder.com/community/data-science/data-science-tutorials/maximum-flow-section-1/>

CSU22012: Data Structures and Algorithms II

Substring Search – part 2

Ivana.Dusparic@scss.tcd.ie

Substring search algorithms – part 2

- › Boyer-Moore
- › Rabin-Karp

So far

- › Brute force
 - $M \times N$ performance
 - Back up
- › KMP
 - $2N$ - linear
 - No backup
 - Extra space – $M \times R$

(N length of a string, M length of a substring we're search for, R radix)

- › can we do better?

Boyer-Moore

Boyer-Moore

- › Big idea – when find a character not in the pattern, can skip up to M characters (so no need to loop through all N characters)
 - Mismatched character heuristic
 - Don't look through characters in order, start from the back and look at the last character in the pattern first and see if it's a match, or in the pattern at all
- › Uses backup

Boyer-Moore

Intuition.

- Scan characters in pattern from right to left.
- Can skip as many as M text chars when finding one not in the pattern.

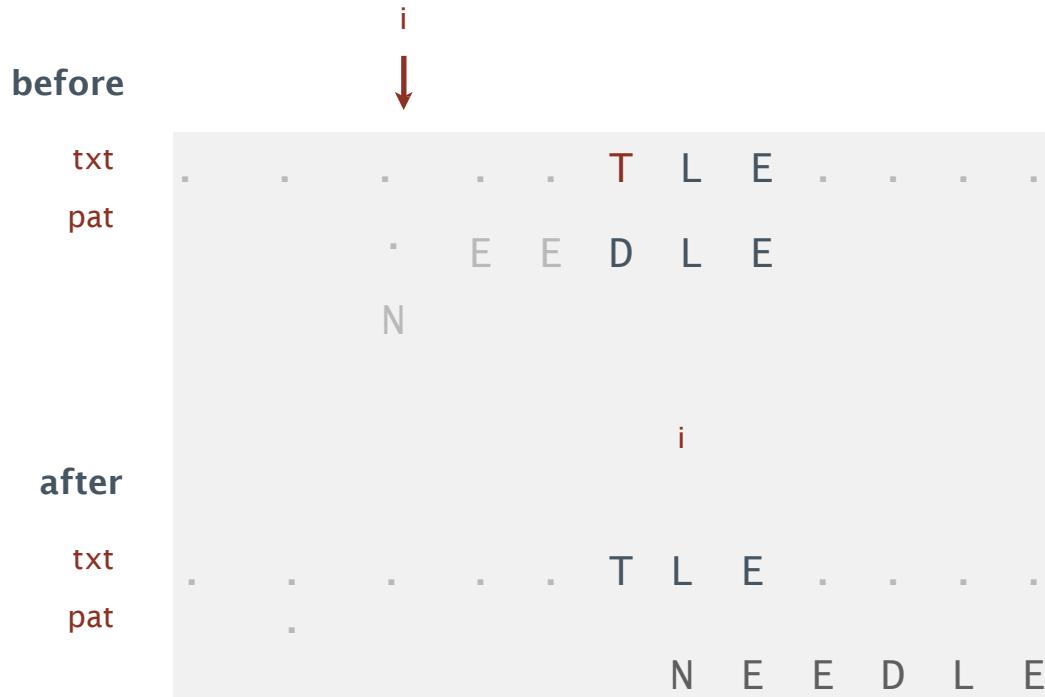
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23		
		F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A		
text →																											
0	5	N	E	E	D	L	E	← pattern																			
5	5								N	E	E	D	L	E													
11	4														N	E	E	D	L	E							
15	0															N	E	E	D	L	E						

return i = 15

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 1. Mismatch character not in pattern.



mismatch character 'T' not in pattern: increment i one character beyond 'T'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2a. Mismatch character in pattern.

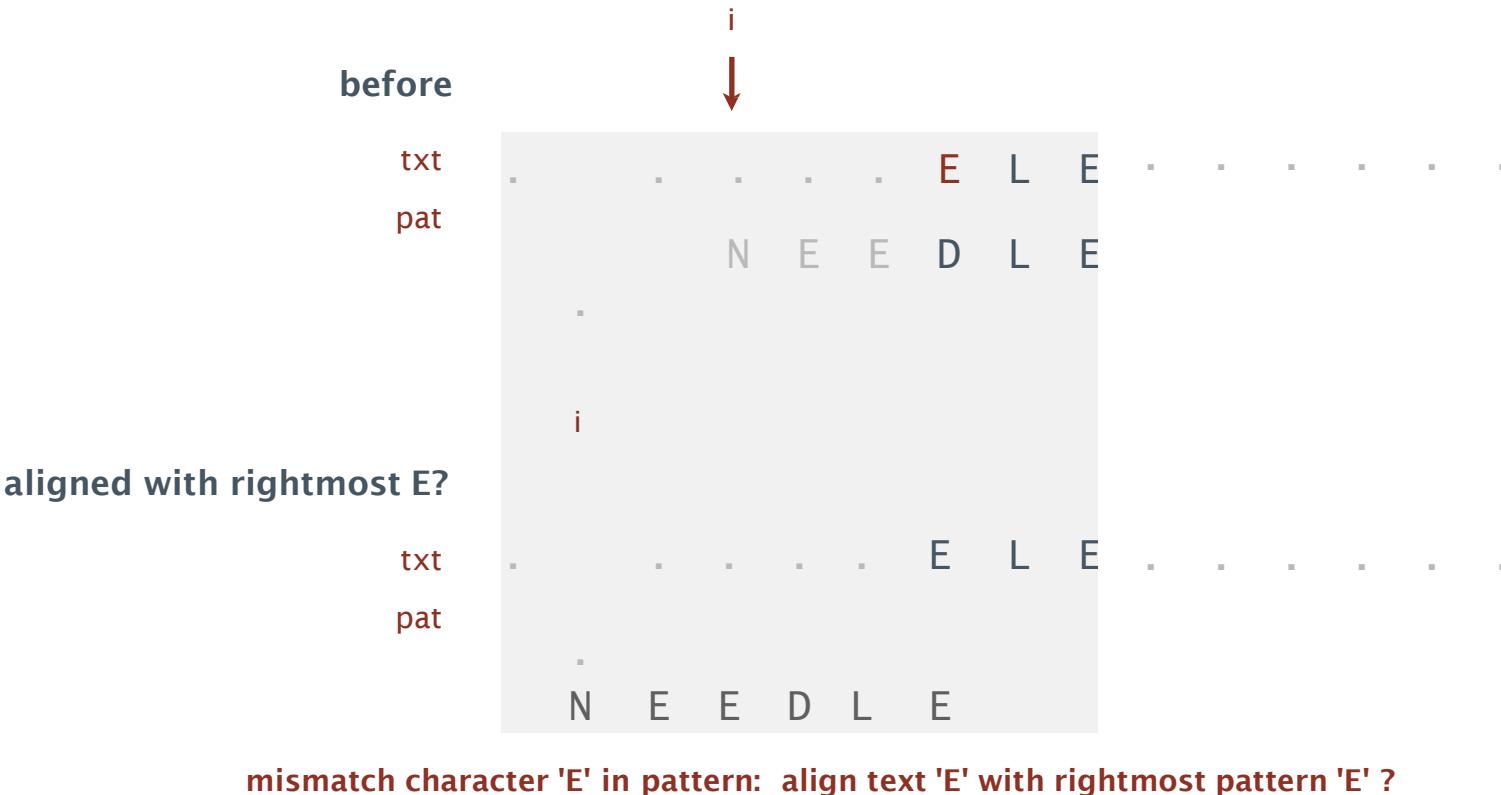


mismatch character 'N' in pattern: align text 'N' with rightmost pattern 'N'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

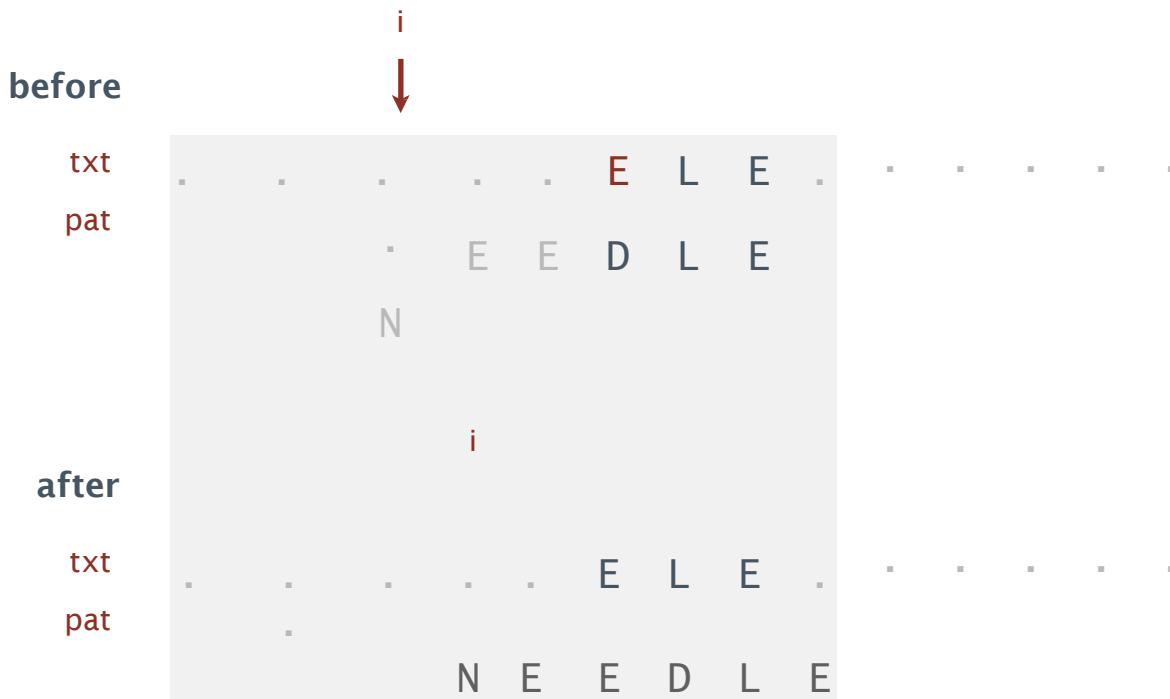
Case 2b. Mismatch character in pattern (but heuristic no help).



Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).



mismatch character 'E' in pattern: increment i by 1

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Precompute index of rightmost occurrence of character c in pattern.
(-1 if character not in pattern)

```
right = new int[R];
for (int c = 0; c < R; c++)
    right[c] = -1;
for (int j = 0; j < M; j++)
    right[pat.charAt(j)] = j;
```

c	N	E	E	D	L	E	right[c]
	0	1	2	3	4	5	
A	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3
E	-1	-1	1	2	2	2	5
...							-1
L	-1	-1	-1	-1	-1	4	4
M	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0
...							-1

Boyer-Moore skip table computation

Precomputing the index of right-most occurrence

```
// position of rightmost occurrence of c in the pattern
right = new int[R];
for (int c = 0; c < R; c++)
    right[c] = -1;
for (int j = 0; j < pattern.length; j++)
    right[pattern[j]] = j;
```

Boyer-Moore: Java implementation

```
public int search(String txt)
{
    int N = txt.length();
    int M = pat.length();
    int skip;
    for (int i = 0; i <= N-M; i += skip)
    {
        skip = 0;
        for (int j = M-1; j >= 0; j--)
        {
            if (pat.charAt(j) != txt.charAt(i+j))
            {
                skip = Math.max(1, j - right[txt.charAt(i+j)]);
                break;
            }
        }
        if (skip == 0) return i; ← match
    }
    return N;
}
```

compute skip value

in case other term is nonpositive

Boyer-Moore exercise

- › Fill in the table containing mismatched character heuristic for the word “test”
- › Write the trace of how many characters you skipped when searching for “somepatterntotestin” in the string, by keeping track of i and j increments/decrements

- › <https://people.ok.ubc.ca/ylucet/DS/BoyerMoore.html>
- › Demo – use this to check your exercise

Boyer-Moore: analysis

Property. Substring search with the Boyer-Moore mismatched character heuristic takes about $\sim N/M$ character compares to search for a pattern of length M in a text of length N .  **sublinear!**

What's the worst case input/performance of Boyle-Moore?

Boyer-Moore: analysis

Worst-case. Can be as bad as $\sim MN$.

i	skip	0	1	2	3	4	5	6	7	8	9
	txt	B	B	B	B	B	B	B	B	B	B
0	0	A	B	B	B	B					pat
1	1		A	B	B	B	B				
2	1			A	B	B	B	B			
3	1				A	B	B	B	B		
4	1					A	B	B	B	B	
5	1						A	B	B	B	B

Rabin-Karp

Rabin-Karp

Basic idea = modular hashing.

- Compute a hash of $\text{pat}[0..M-1]$.
 - For each i , compute a hash of $\text{txt}[i..M+i-1]$.
 - If pattern hash = text substring hash, check for a match.

pat.charAt(i)															
i	0	1	2	3	4										
	2	6	5	3	5	%	997	=	613						
txt.charAt(i)															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	
0	3	1	4	1	5	%	997	=	508						
1		1	4	1	5	9	%	997	=	201					
2			4	1	5	9	2	%	997	=	715				
3				1	5	9	2	6	%	997	=	971			
4					5	9	2	6	5	%	997	=	442		
5						9	2	6	5	3	%	997	=	929	
6 ← return i = 6							2	6	5	3	5	%	997	=	613

modular hashing with $R = 10$ and $\text{hash}(s) = s \pmod{997}$

Modular hashing

- › Remember hash tables
- › *hash function* - map data of arbitrary size to data of fixed size
- › Eg map any value to an index in the array
- › With **modular hashing**, the hash function is simply $h(k) =$
- › $k \bmod m$ for some m . The value k is an integer hash code generated from the key (generally used with positive integers)

```
int h(int k, int M) {  
    return k% M;  
}
```

Rabin-Karp – modular hashing

› Won't it overflow, for large search substrings?

Math trick. To keep numbers small, take intermediate results modulo Q .

Ex.

$$\begin{aligned}(10000 + 535) * 1000 &\pmod{997} \\ = (30 + 535) * 3 &\pmod{997} \\ = 1695 &\pmod{997} \\ = 698 &\pmod{997}\end{aligned}$$

$$(a + b) \bmod Q = ((a \bmod Q) + (b \bmod Q)) \bmod Q$$

$$(a * b) \bmod Q = ((a \bmod Q) * (b \bmod Q)) \bmod Q$$

two useful modular arithmetic identities

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0 \pmod{Q}$$

Intuition. M -digit, base- R integer, modulo Q .

Horner's method. Linear-time method to evaluate degree- M polynomial.

pat.charAt()					
i	0	1	2	3	4
	2	6	5	3	5
0	2	% 997	= 2		
1	2	6	% 997 = (2*10 + 6) % 997 = 26	$\nearrow R$	$\searrow Q$
2	2	6	5 % 997 = (26*10 + 5) % 997 = 265		
3	2	6	5 3 % 997 = (265*10 + 3) % 997 = 659		
4	2	6	5 3 5 % 997 = (659*10 + 5) % 997 = 613		

```
// Compute hash for M-digit key
private long hash(String key, int M)
{
    long h = 0;
    for (int j = 0; j < M; j++)
        h = (h * R + key.charAt(j)) % Q;
    return h;
}
```

$$\begin{aligned}26535 &= 2*10000 + 6*1000 + 5*100 + 3*10 + 5 \\&= (((2 * 10 + 6) * 10 + 5) * 10 + 3) * 10 + 5\end{aligned}$$

Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i .

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0$$

$$x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \dots + t_{i+M} R^0$$

Key property. Can update "rolling" hash function in constant time!

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M}$$

↑ ↑ ↑ ↑
current subtract multiply add new
value leading digit by radix trailing digit
(can precompute R^{M-1})

i	...	2	3	4	5	6	7	...
current value		1	4	1	5	9	2	6
new value		4	1	5	9	2	6	5

$$\begin{array}{r} 4 & 1 & 5 & 9 & 2 & \text{current value} \\ - 4 & 0 & 0 & 0 & 0 & \\ 1 & 5 & 9 & 2 & & \text{subtract leading digit} \\ * & 1 & 0 & & & \text{multiply by radix} \\ 1 & 5 & 9 & 2 & 0 & \\ & & & + & 6 & \text{add new trailing digit} \\ 1 & 5 & 9 & 2 & 6 & \text{new value} \end{array}$$

Rabin-Karp substring search example

First R entries: Use Horner's rule.

Remaining entries: Use rolling hash (and % to avoid overflow).

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	7	9	3	
0	3	1	% 997 = 3								Q	9				
1	3	1	% 997 = (3*10 + 1) % 997 = 31													
2	3	1	4	% 997 = (31*10 + 4) % 997 = 314												
3	3	1	4	1	% 997 = (314*10 + 1) % 997 = 150											
4	3	1	4	1	5	% 997 = (150*10 + 5) % 997 = 508	RM	R								
5	1	4	1	5	9	% 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201										
6		4	1	5	9	2	% 997 = ((201 + 1*(997 - 30))*10 + 2) % 997 = 715									
7			1	5	9	2	6	% 997 = ((715 + 4*(997 - 30))*10 + 6) % 997 = 971								
8				5	9	2	6	5	% 997 = ((971 + 1*(997 - 30))*10 + 5) % 997 = 442	match						
9					9	2	6	5	3	% 997 = ((442 + 5*(997 - 30))*10 + 3) % 997 = 929						
10	← return $i-M+1 = 6$	2	6	5	3	5	5	% 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613								

-30 (mod 997) = 997 - 30 10000 (mod 997) = 30

Horner's rule

rolling hash

match

Rabin-Karp: Java implementation

```
public class RabinKarp
{
    private long patHash;      // pattern hash
                                value
    private int M;             // pattern length
    private long Q;            // modulus
    private int R;              / radix
    private long RM1;           /  $R^{M-1} \bmod Q$ 
    public RabinKarp(String pat) {
        M = pat.length();       /
        R = 256;
        Q = longRandomPrime();   ← a large prime
                                (but avoid overflow)

        RM1 = 1;
        for (int i = 1; i <= M-1; i++)
            RM1 = (R * RM1) % Q;
        patHash = hash(pat, M);   ← precompute  $R^{M-1} \bmod Q$ 
    }

    private long hash(String key, int M)
    { /* as before */ }

    public int search(String txt)
    { /* see next slide */ }
}
```

Rabin-Karp: Java implementation (continued)

Monte Carlo version. Return match if hash match.

```
public int search(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

check for hash collision
using rolling hash function

Las Vegas version. Check for substring match if hash match;
continue search if false collision.



Rabin-Karp analysis

Monte Carlo version.

- Always runs in linear time.
- Extremely likely to return correct answer (but not always!).

Las Vegas version.

- Always returns correct answer.
- Extremely likely to run in linear time (but worst case is MN).

Rabin-Karp

Advantages.

- Extends to 2d patterns.
- Extends to finding multiple patterns.

Disadvantages.

- Arithmetic ops slower than char compares.
- Las Vegas version requires backup.
- Poor worst-case guarantee.

Substring search cost summary

Cost of searching for an M -character pattern in an N -character text.

algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1N$	yes	yes	I
Knuth-Morris-Pratt	<i>full DFA (Algorithm 5.6)</i>	$2N$	$1.1N$	no	yes	MR
	<i>mismatch transitions only</i>	$3N$	$1.1N$	no	yes	M
Boyer-Moore	<i>full algorithm</i>	$3N$	N/M	yes	yes	R
	<i>mismatched char heuristic only (Algorithm 5.7)</i>	MN	N/M	yes	yes	R
Rabin-Karp[†]	<i>Monte Carlo (Algorithm 5.8)</i>	$7N$	$7N$	no	yes^t	I
	<i>Las Vegas</i>	$7N^t$	$7N$	yes	yes	I

^t probabilistic guarantee, with uniform hash function

So which algorithm should I use?

- › Java String.contains() method – brute force
 - Very compact, very little operations inside the loop, so loop runs fast.
As there is no overhead it performs better when searching short strings.
- › Boyer-Moore
 - grep
 - Works well for long search patterns
 - The more distinct letters in the strings, the better the positive impact
 - backup
- › KMP
 - Small alphabet, repeated subpatterns
 - No backup

CSU22012: Data Structures and Algorithms II

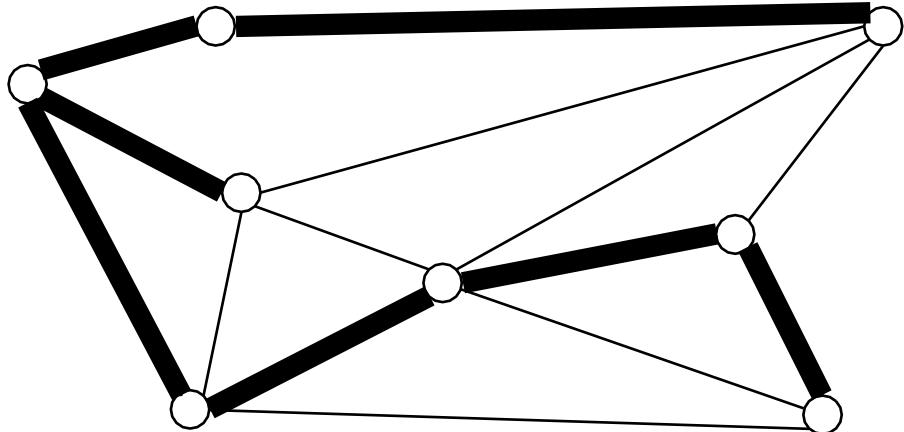
Minimum Spanning Trees

Ivana.Dusparic@scss.tcd.ie

Spanning tree

A **spanning tree** of G is a subgraph T that is:

- Connected.
- Acyclic.
- Includes all of the vertices.

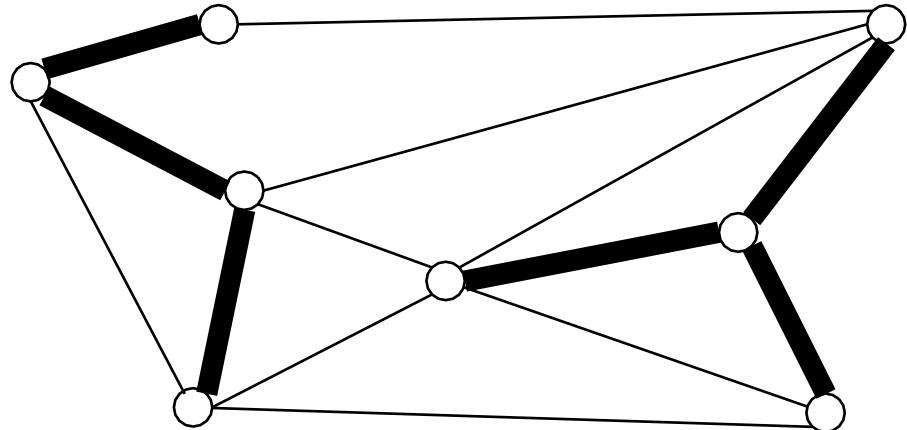


graph G

Spanning tree

Def. A **spanning tree** of G is a subgraph T that is:

- Connected.
- Acyclic.
- Includes all of the vertices.

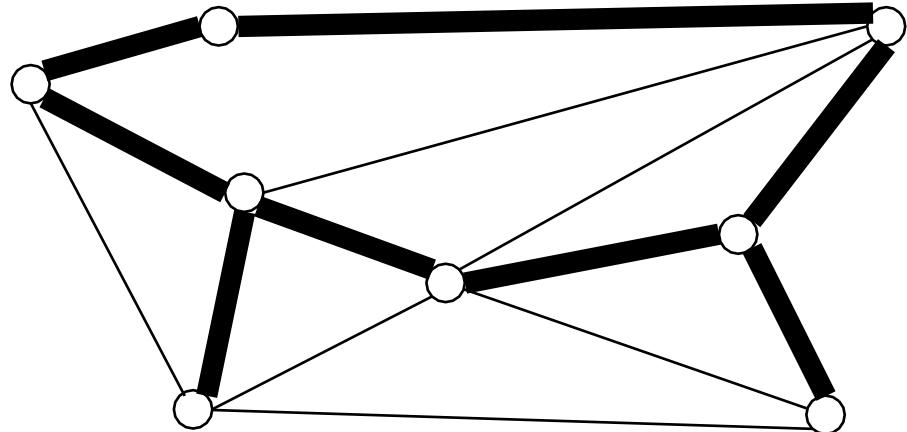


not connected

Spanning tree

Def. A **spanning tree** of G is a subgraph T that is:

- Connected.
- Acyclic.
- Includes all of the vertices.

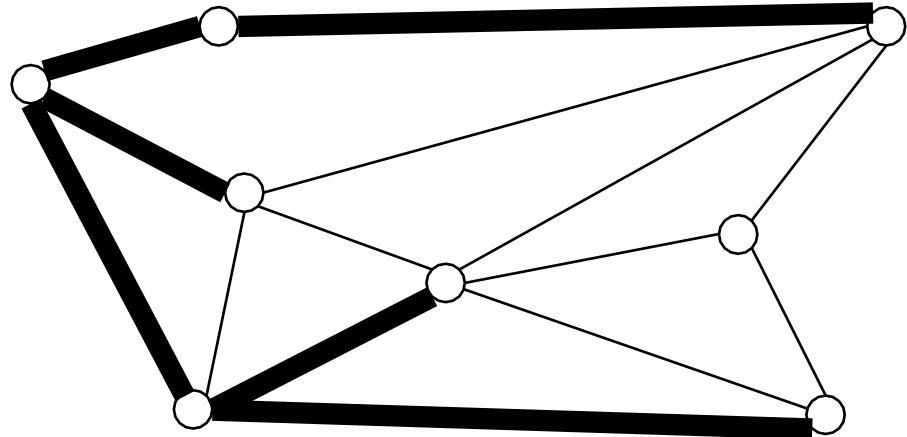


not acyclic

Spanning tree

Def. A **spanning tree** of G is a subgraph T that is:

- Connected.
- Acyclic.
- Includes all of the vertices.

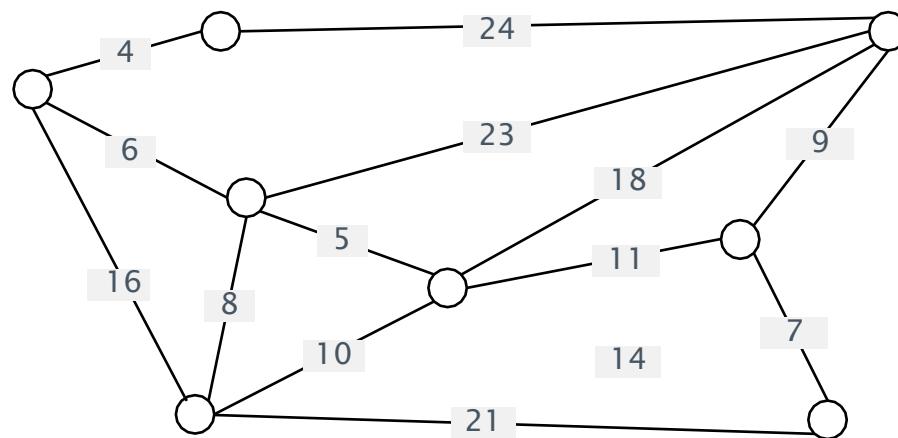


not spanning

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Goal. Find a min weight spanning tree.

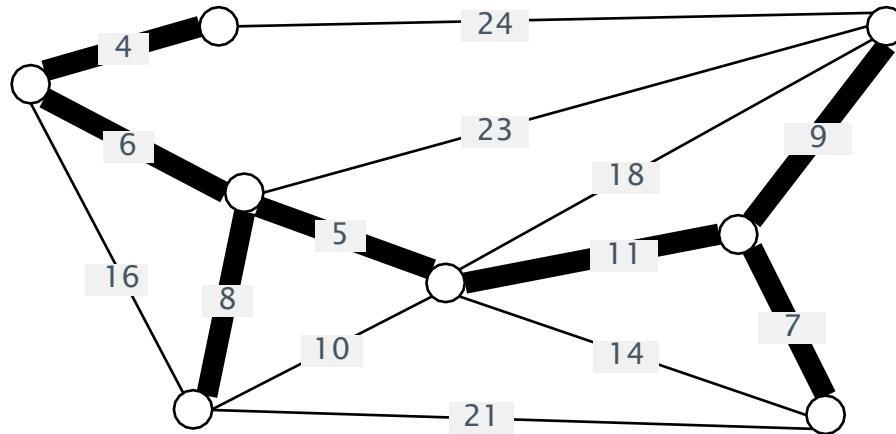


edge-weighted graph G

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Goal. Find a min weight spanning tree.



minimum spanning tree T
(cost = $50 = 4 + 6 + 8 + 5 + 11 + 9 + 7$)

Brute force. Try all spanning trees?

Applications

- › Network design (communication, electrical, hydraulic, computer, road etc)
- › Clustering
- › Approximation algorithms (eg TSP)
- › Many others – classification in biology, sociology, face verification, hand writing detection etc

MST question – Turning Point

- › Let G be a connected, edge-weighted graph with V vertices and E edges. How many edges are in a minimum spanning tree of G ?
 - V
 - $V-1$
 - E
 - $E-1$

Algorithms

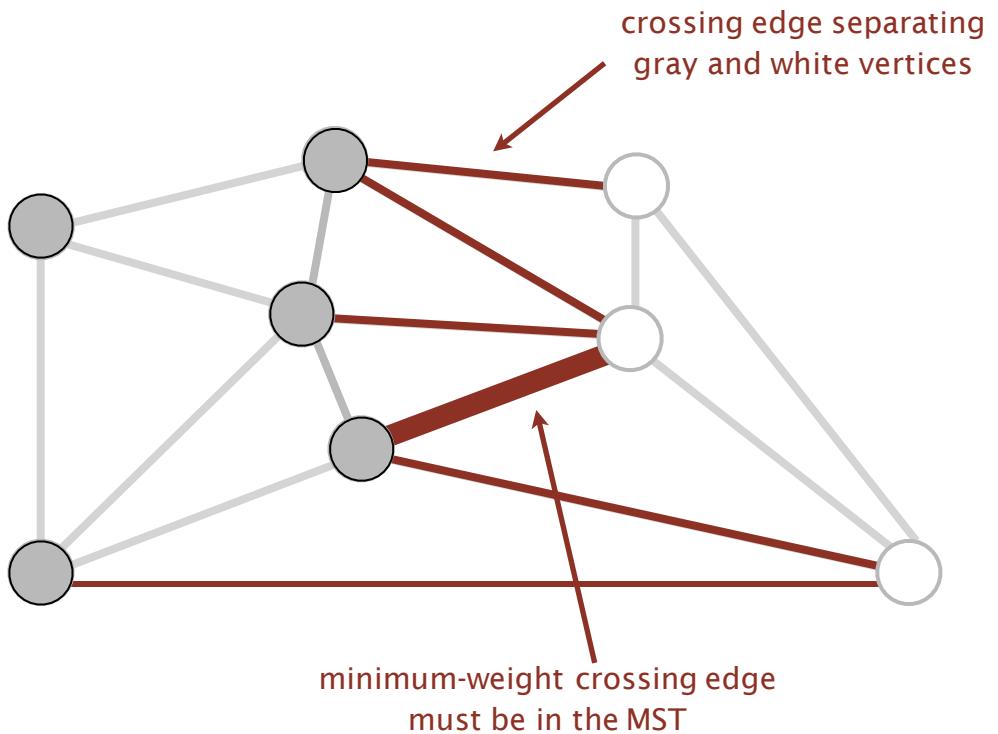
- › Greedy algorithms
 - Prim's
 - Kruskal's

Cut property

Def. A **cut** in a graph is a partition of its vertices into two (nonempty) sets.

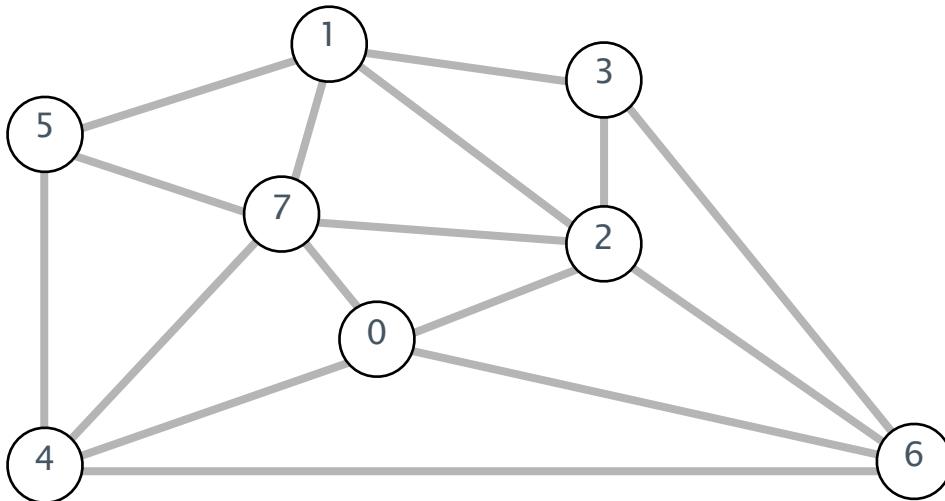
Def. A **crossing edge** connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.



Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

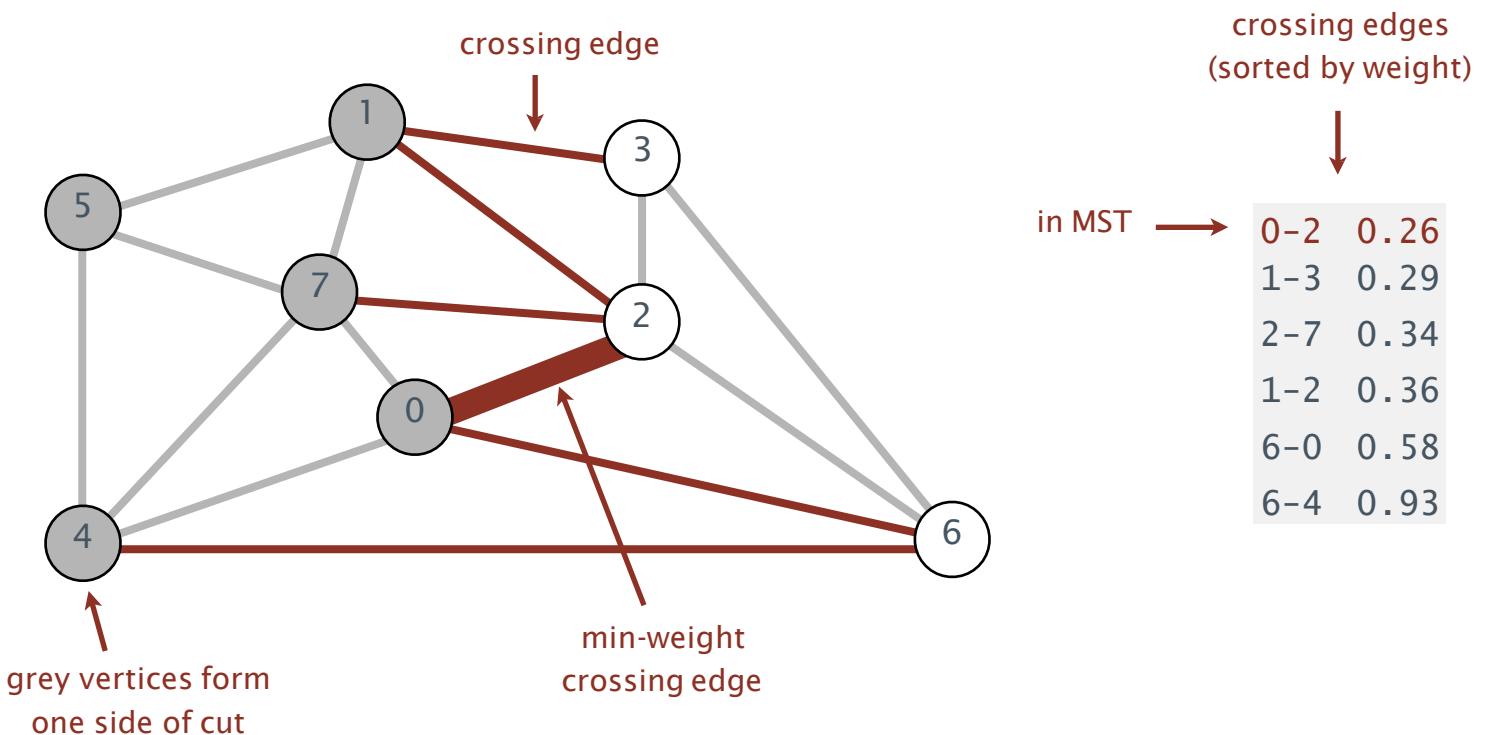


an edge-weighted graph

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

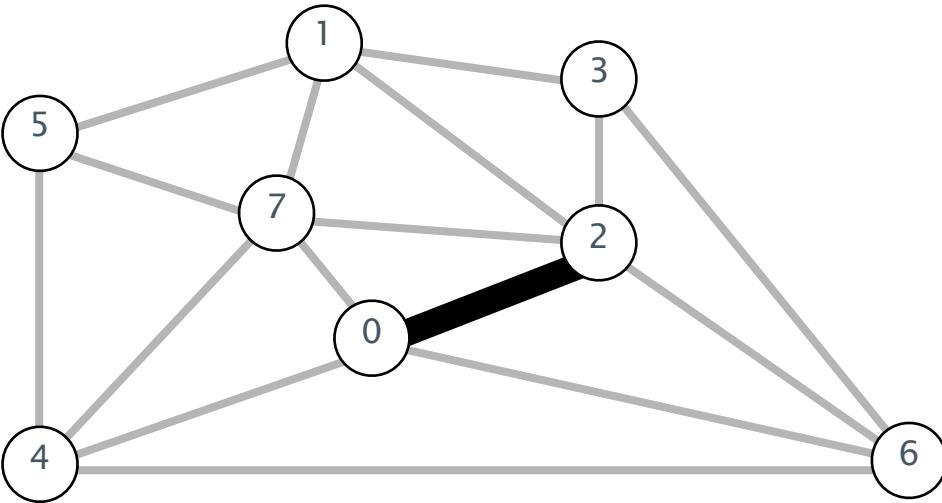
Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

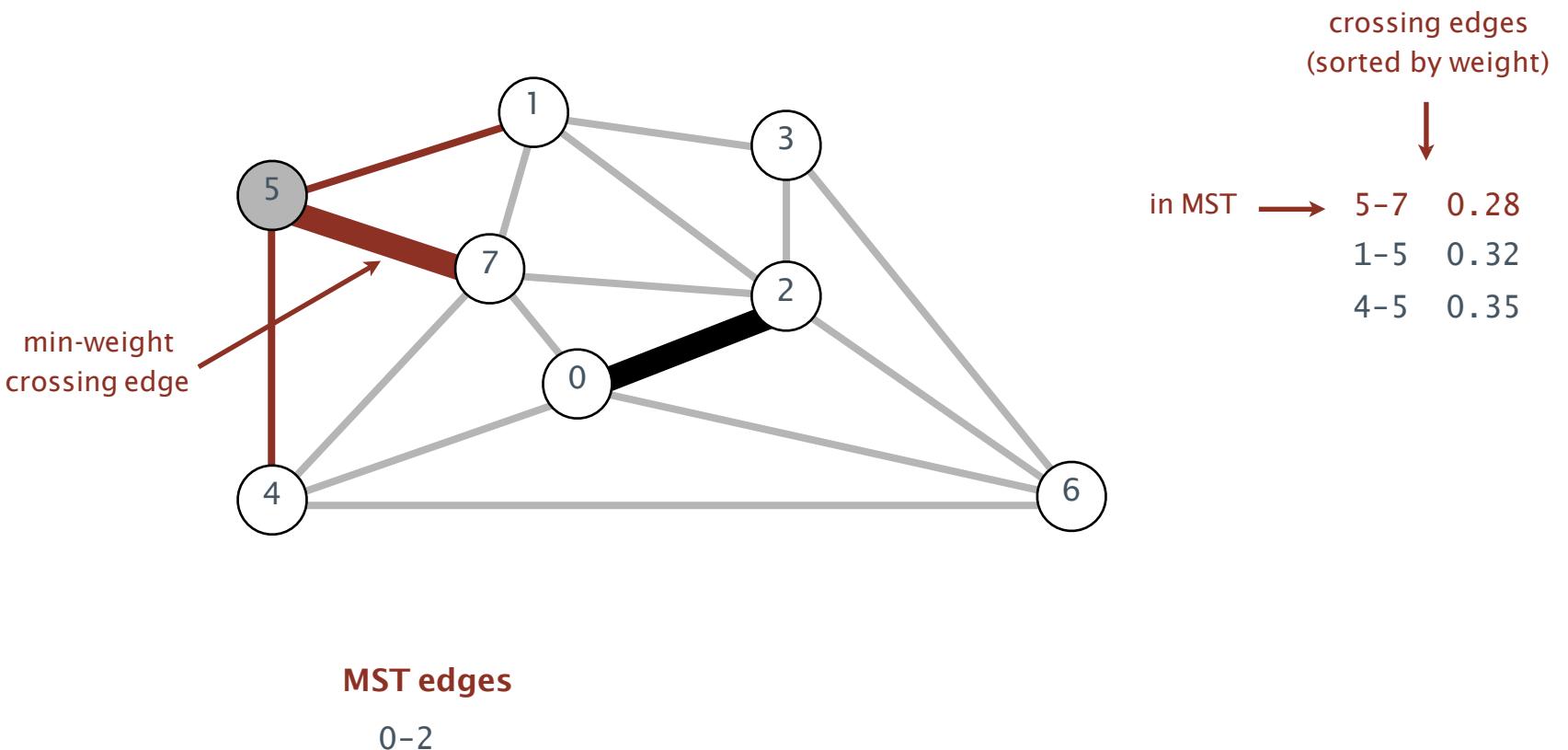


MST edges

0-2

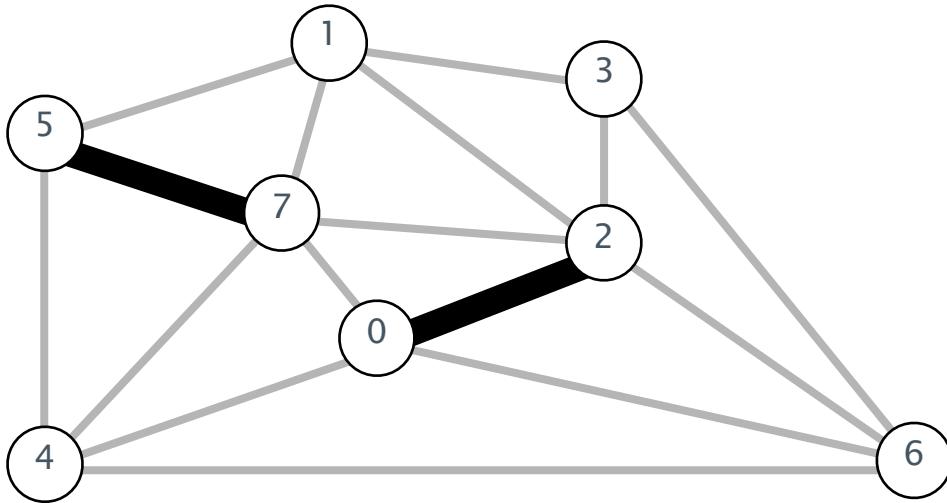
Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

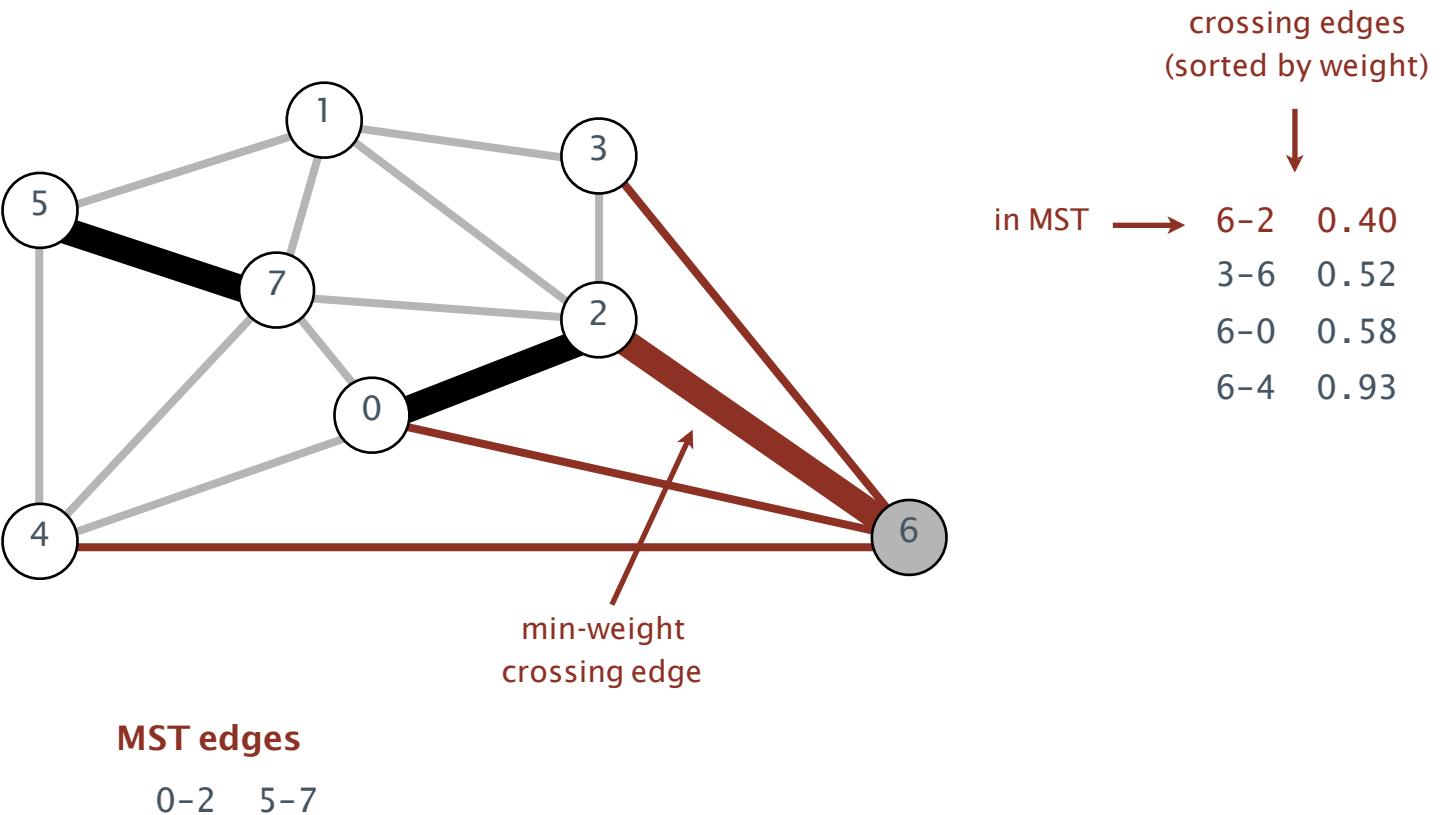


MST edges

0-2 5-7

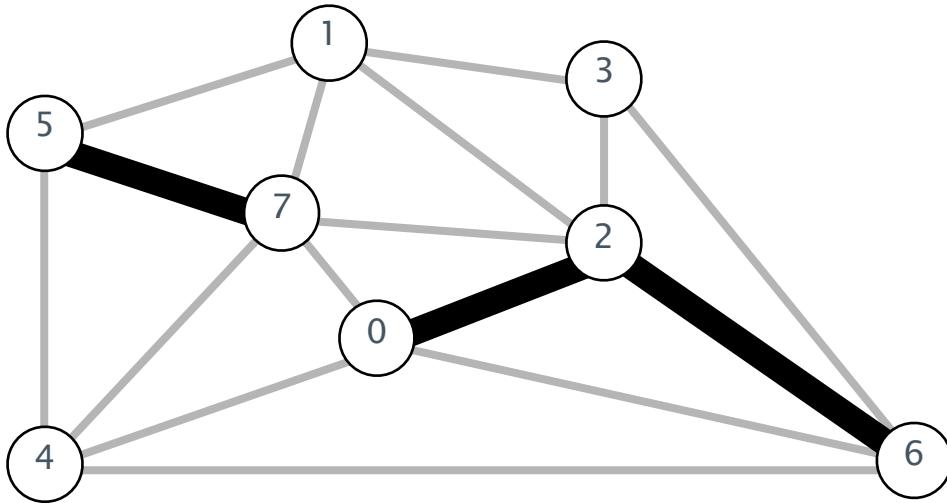
Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

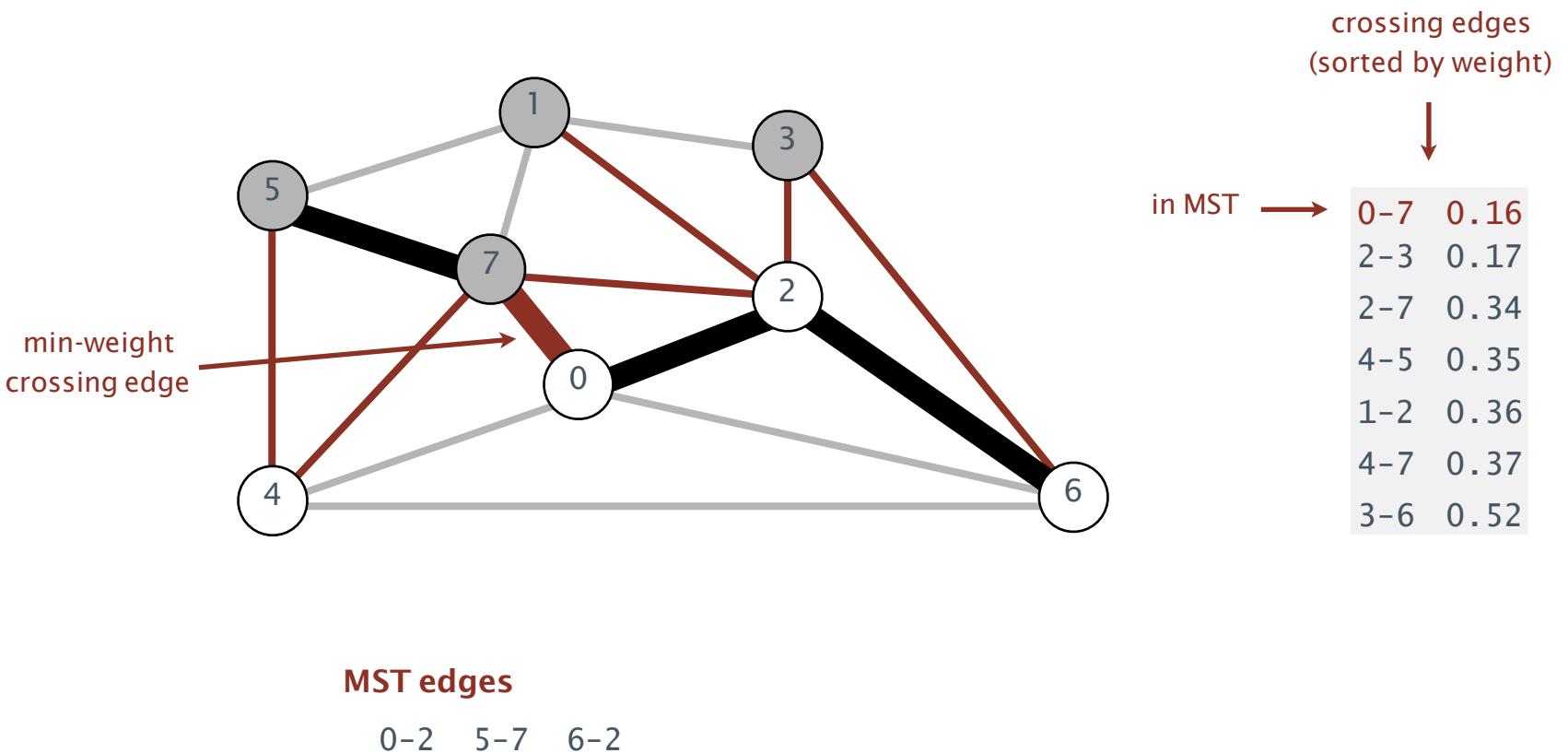


MST edges

0-2 5-7 6-2

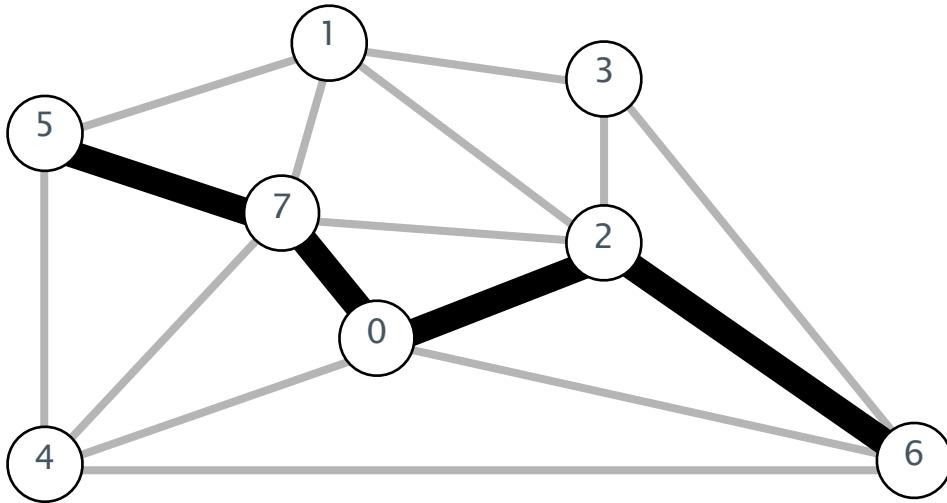
Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

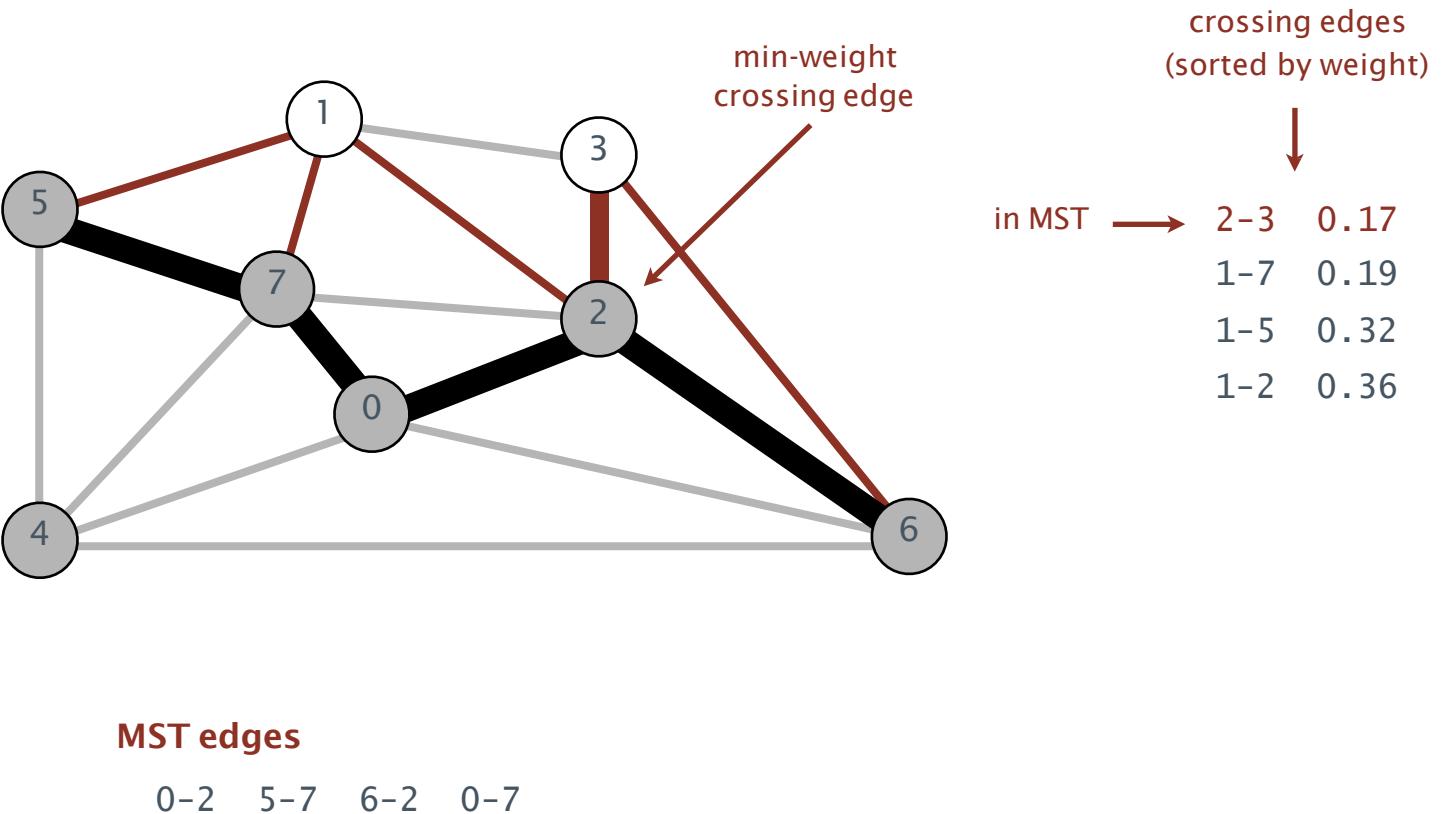


MST edges

0-2 5-7 6-2 0-7

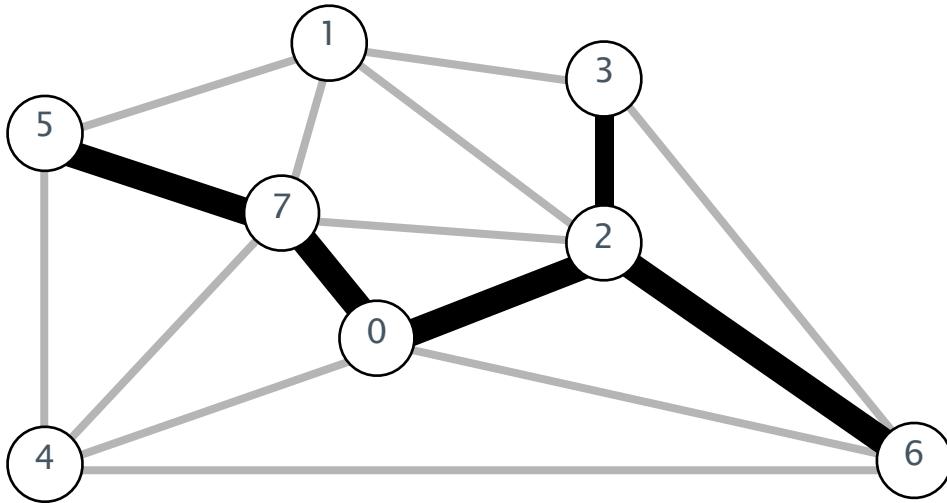
Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

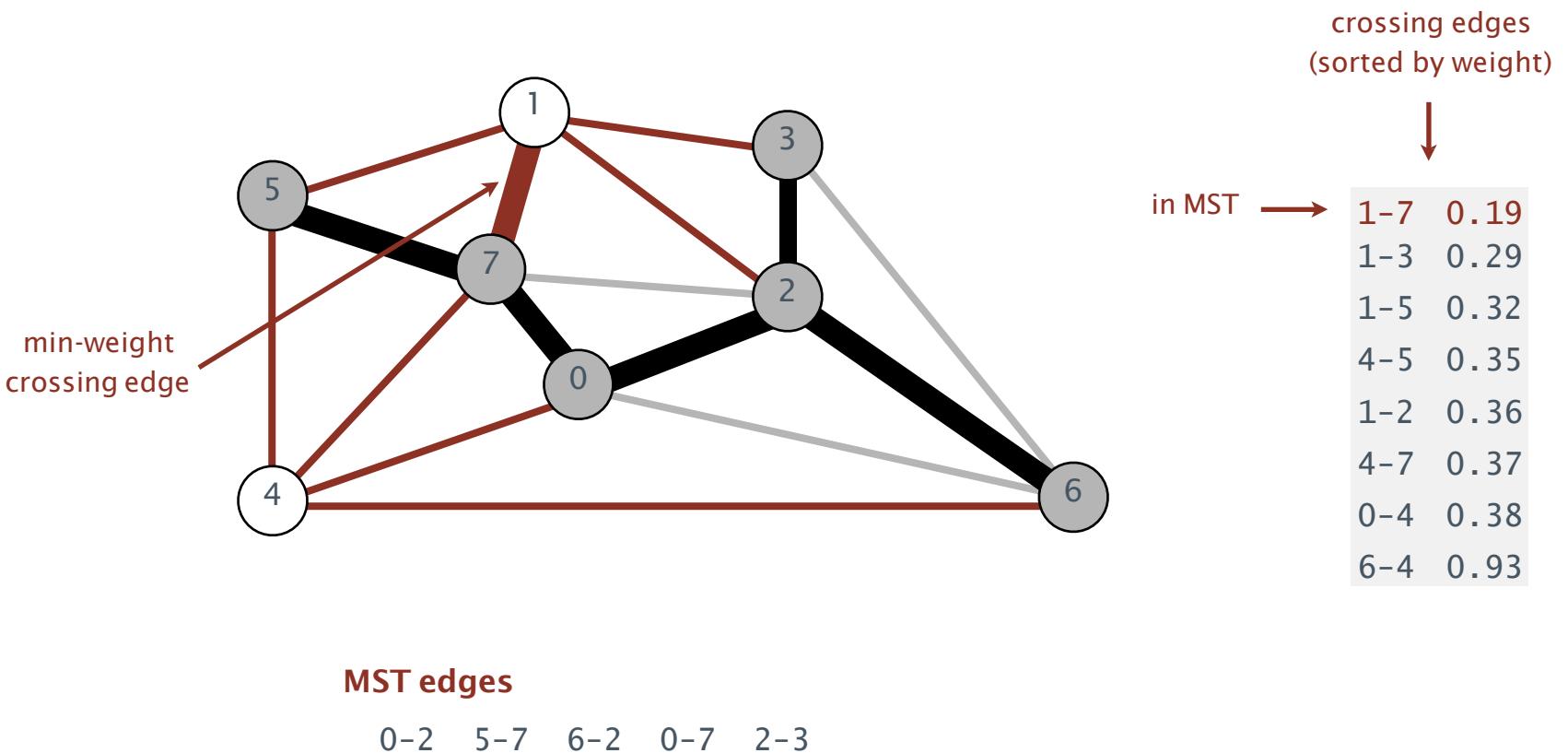


MST edges

0-2 5-7 6-2 0-7 2-3

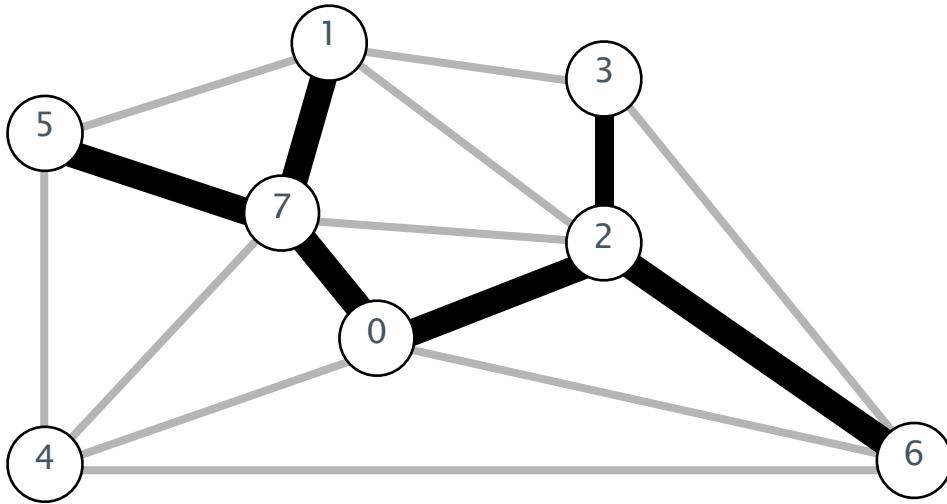
Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

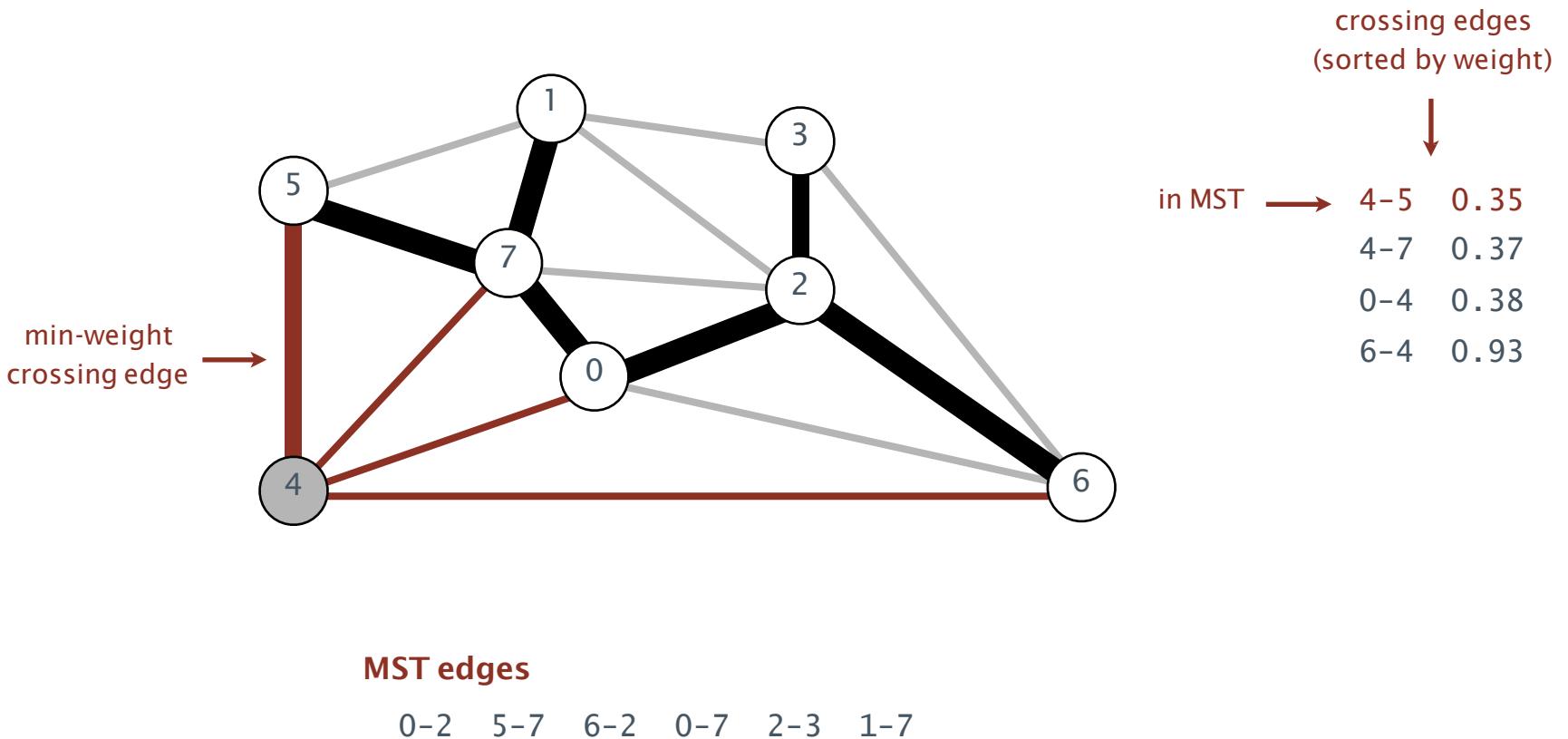


MST edges

0-2 5-7 6-2 0-7 2-3 1-7

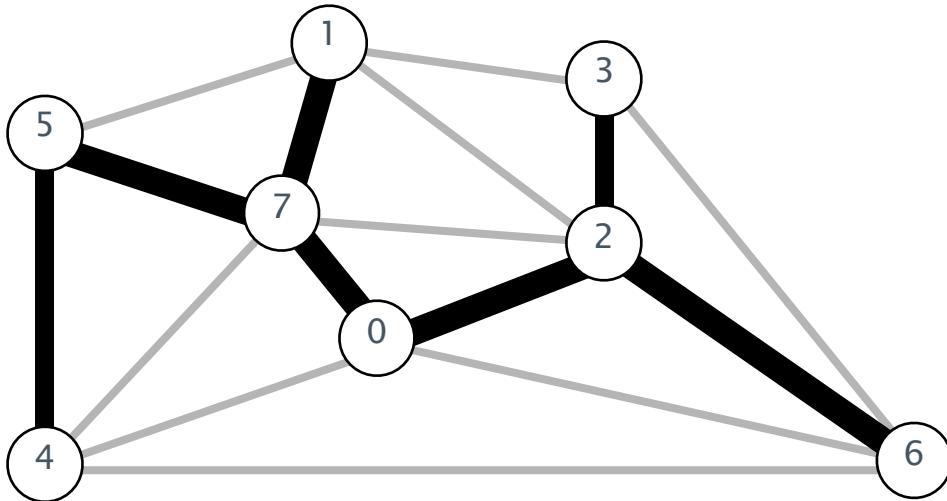
Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



MST edges

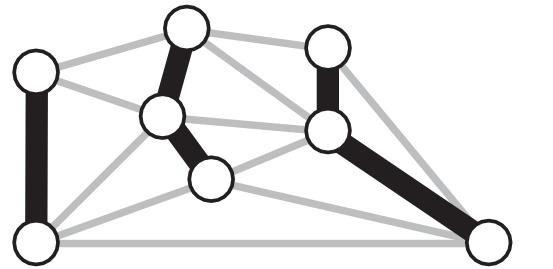
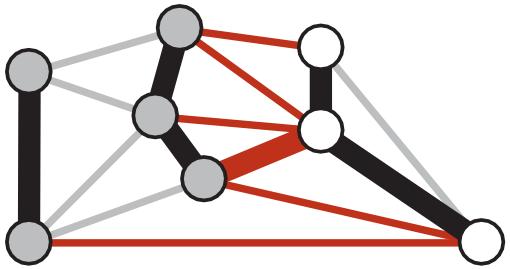
0-2 5-7 6-2 0-7 2-3 1-7 4-5

Greedy MST algorithm: correctness proof

Proposition. The greedy algorithm computes the MST.

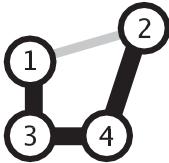
Pf.

- Any edge colored black is in the MST (via cut property).
- Fewer than $V - 1$ black edges \Rightarrow cut with no black crossing edges.
(consider cut whose vertices are any one connected component)

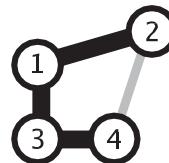


Q. What if edge weights are not all distinct?

A. Greedy MST algorithm still correct if equal weights are present!
(our correctness proof fails, but that can be fixed)



1	2	1.00
1	3	0.50
2	4	1.00
3	4	0.50

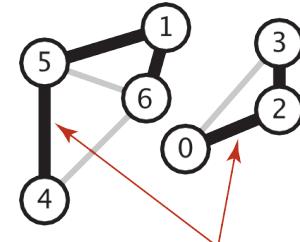


1	2	1.00
1	3	0.50
2	4	1.00
3	4	0.50

4	5	0.6
	1	
4	6	0.6
	2	
5	6	0.8
	8	
1	5	0.1
	1	
2	3	0.3
	5	
0	3	0.6
	1	
1	6	0.1
	0	
0	2	0.2
	2	

Q. What if graph is not connected?

A. Compute minimum spanning forest = MST of each component.



*can independently compute
MSTs of components*

Greedy MST algorithm: efficient implementations

Efficient implementations.

How to choose cut?

How to find min-weight edge?

Ex 1. Kruskal's algorithm.

Ex 2. Prim's algorithm.

Ex 3. Borüvka's algorithm.

Weighted-edge graph API, Edge API

Adjacency-lists graph representation (review): Java implementation

```
public class Graph
{
    private final int V;
    private final Bag<Integer>[] adj; ← adjacency lists

    public Graph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) ← add edge v-w
    {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) ← iterator for vertices
    {   return adj[v];   } adjacent to v
}
```

Edge-weighted graph: adjacency-lists implementation

```
public class EdgeWeightedGraph
{
    private final int V;
    private final Bag<Edge>[] adj;

    public EdgeWeightedGraph(int V)
    {
        this.V = V;
        adj = (Bag<Edge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Edge>();
    }

    public void addEdge(Edge e)
    {
        int v = e.either(), w = e.other(v);
        adj[v].add(e);
        adj[w].add(e);
    }

    public Iterable<Edge> adj(int v)
    {
        return adj[v];
    }
}
```

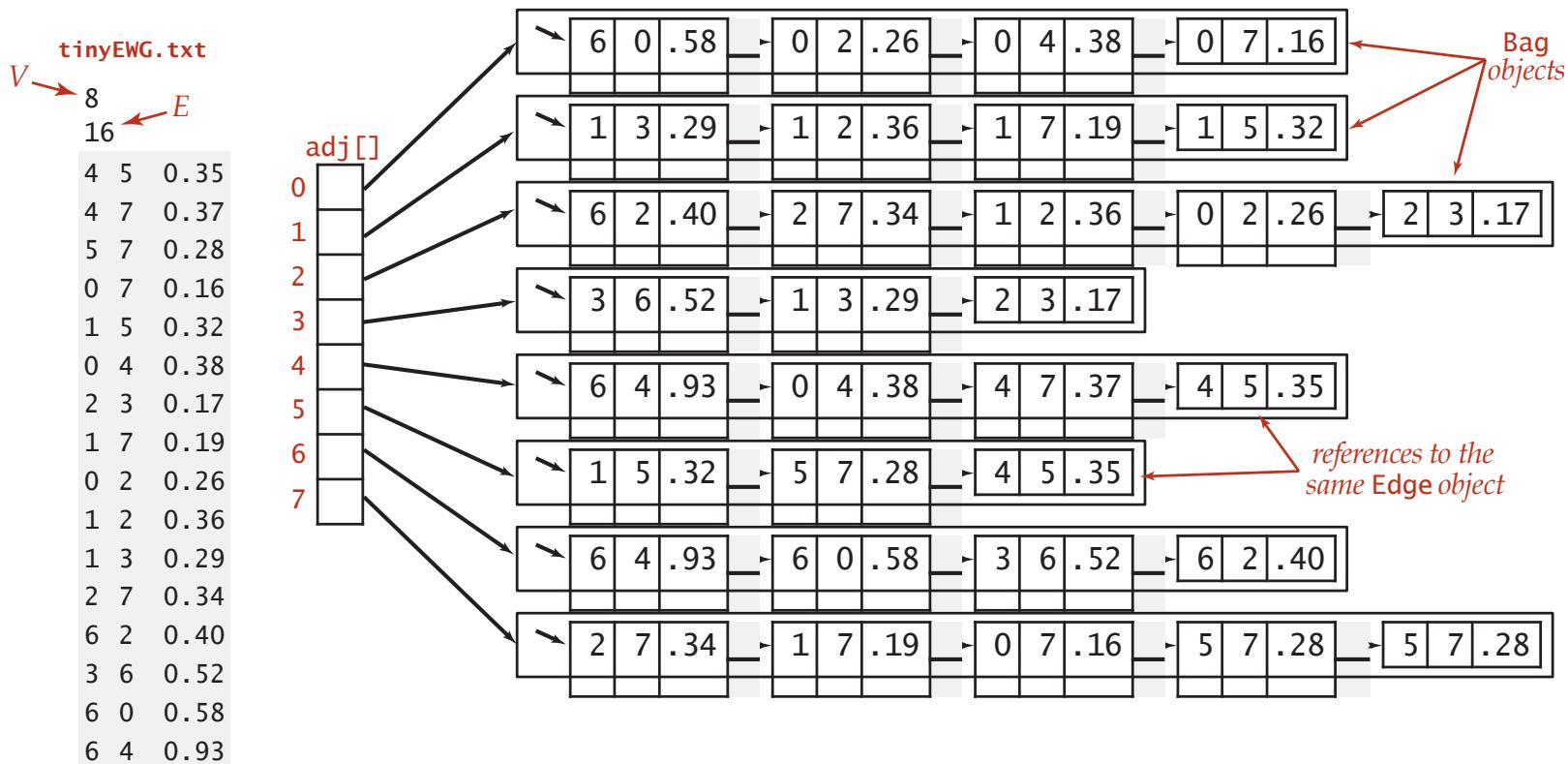
same as Graph, but adjacency lists of Edges instead of integers

constructor

add edge to both adjacency lists

Edge-weighted graph: adjacency-lists representation

Maintain vertex-indexed array of Edge lists.



Edge-weighted graph API

```
public class EdgeWeightedGraph
```

```
    EdgeWeightedGraph(int V)
```

create an empty graph with V vertices

```
    EdgeWeightedGraph(In in)
```

create a graph from input stream

```
    void addEdge(Edge e)
```

add weighted edge e to this graph

```
    Iterable<Edge> adj(int v)
```

edges incident to v

```
    Iterable<Edge> edges()
```

all edges in this graph

```
    int V()
```

number of vertices

```
    int E()
```

number of edges

```
    String toString()
```

string representation

Weighted Edge API

Edge abstraction needed for weighted edges.

```
public class Edge implements Comparable<Edge>

    Edge(int v, int w, double weight)      create a weighted edge v-w

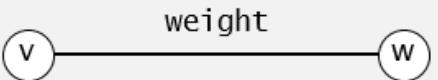
    int either()                          either endpoint

    int other(int v)                     the endpoint that's not v

    int compareTo(Edge that)            compare this edge to that edge

    double weight()                    the weight

    String toString()                  string representation
```



Idiom for processing an edge e: `int v = e.either(), w = e.other(v);`

Weighted edge: Java implementation

```
public class Edge implements Comparable<Edge>
{
    private final int v, w;
    private final double weight;

    public Edge(int v, int w, double weight)
    {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int either()
    {   return v;   }

    public int other(int vertex)
    {
        if (vertex == v) return w;
        else return v;
    }

    public int compareTo(Edge that)
    {
        if      (this.weight < that.weight) return -1;
        else if (this.weight > that.weight) return +1;
        else
            return 0;
    }
}
```

constructor

either endpoint

other endpoint

compare edges by weight

MST API

Minimum spanning tree API

Q. How to represent the MST?

```
public class MST
```

```
    MST(EdgeWeightedGraph G)
```

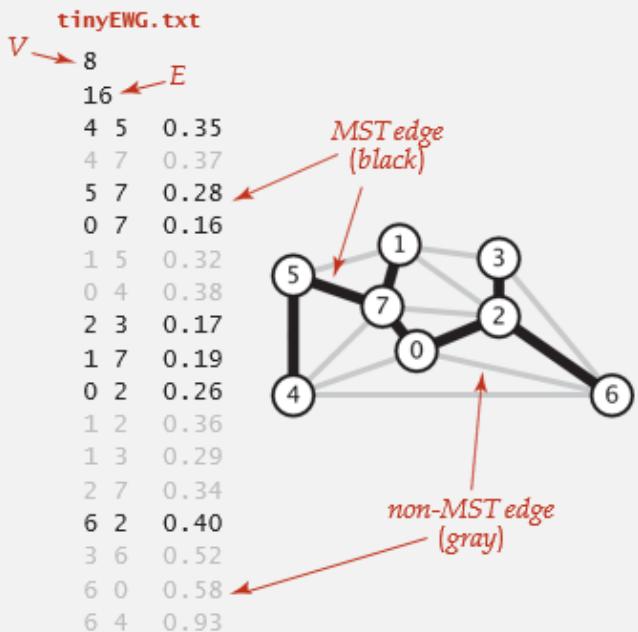
constructor

```
    Iterable<Edge> edges()
```

edges in MST

```
    double weight()
```

weight of MST



```
% java MST tinyEWG.txt
```

```
0-7 0.16  
1-7 0.19  
0-2 0.26  
2-3 0.17  
5-7 0.28  
4-5 0.35  
6-2 0.40  
1.81
```

Minimum spanning tree API

Q. How to represent the MST?

```
public class MST
```

```
    MST(EdgeWeightedGraph G)
```

constructor

```
    Iterable<Edge> edges()
```

edges in MST

```
    double weight()
```

weight of MST

```
public static void main(String[] args)
{
    In in = new In(args[0]);
    EdgeWeightedGraph G = new EdgeWeightedGraph(in);
    MST mst = new MST(G);
    for (Edge e : mst.edges())
        StdOut.println(e);
    StdOut.printf("%.2f\n", mst.weight());
}
```

```
% java MST tinyEWG.txt
```

0-7 0.16

1-7 0.19

0-2 0.26

2-3 0.17

5-7 0.28

4-5 0.35

6-2 0.40

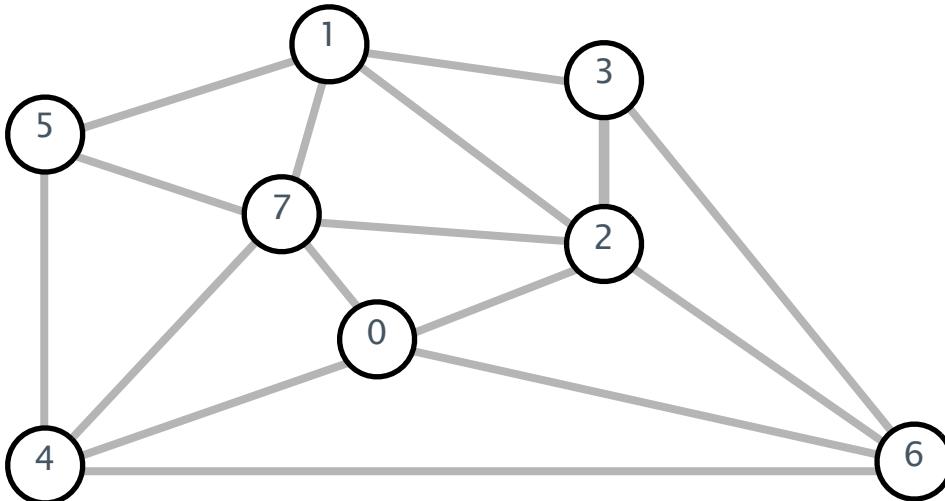
1.81

Kruskal's algorithm

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree T unless doing so would create a cycle.



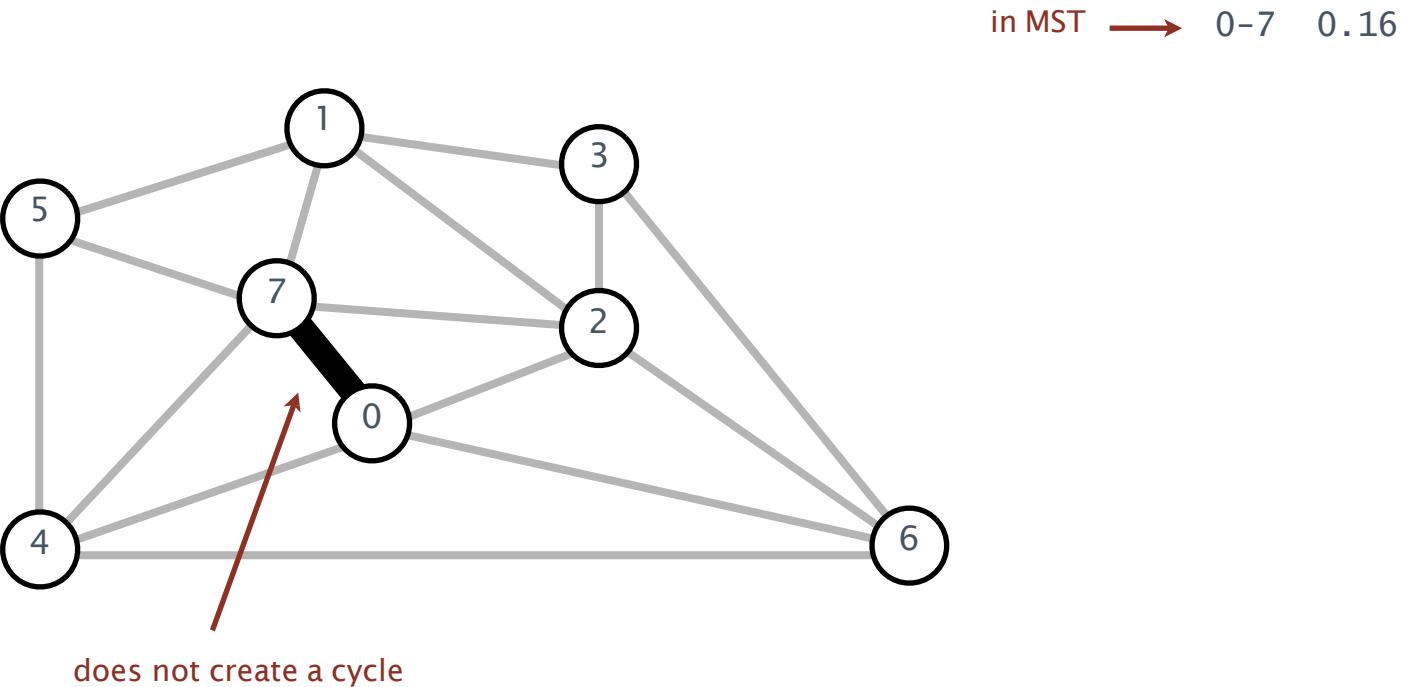
graph edges
sorted by weight

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Kruskal's algorithm demo

Consider edges in ascending order of weight.

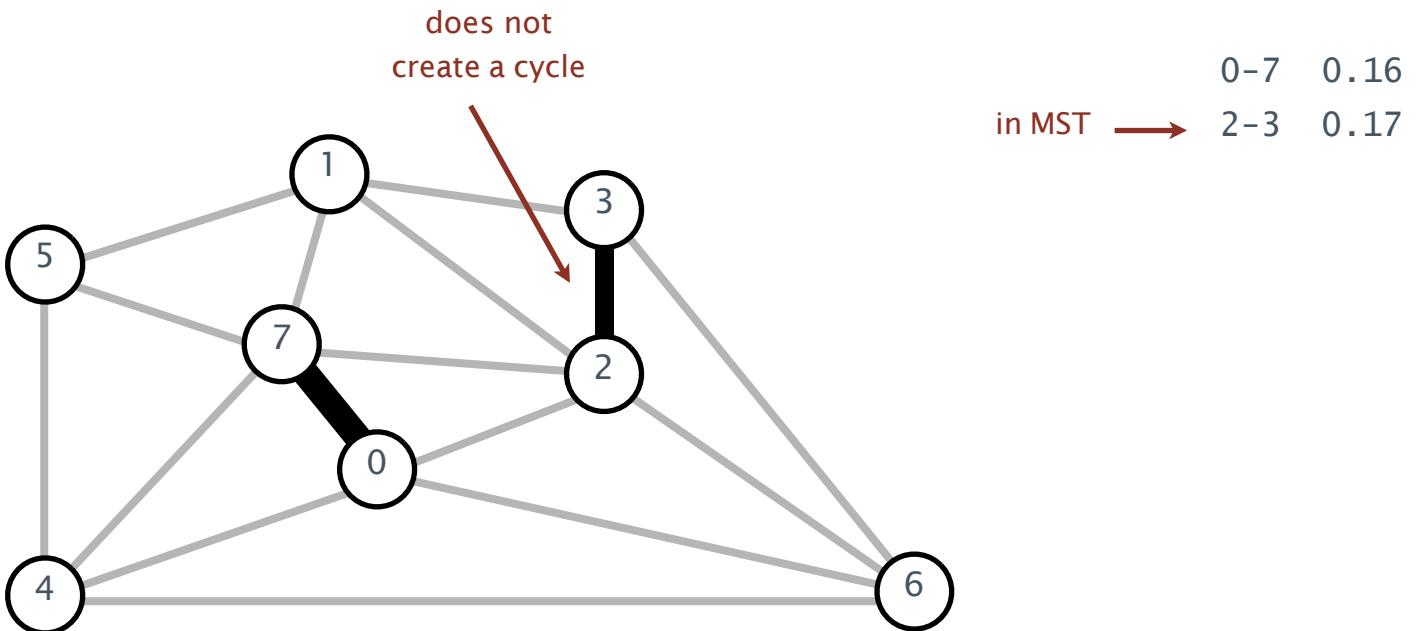
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

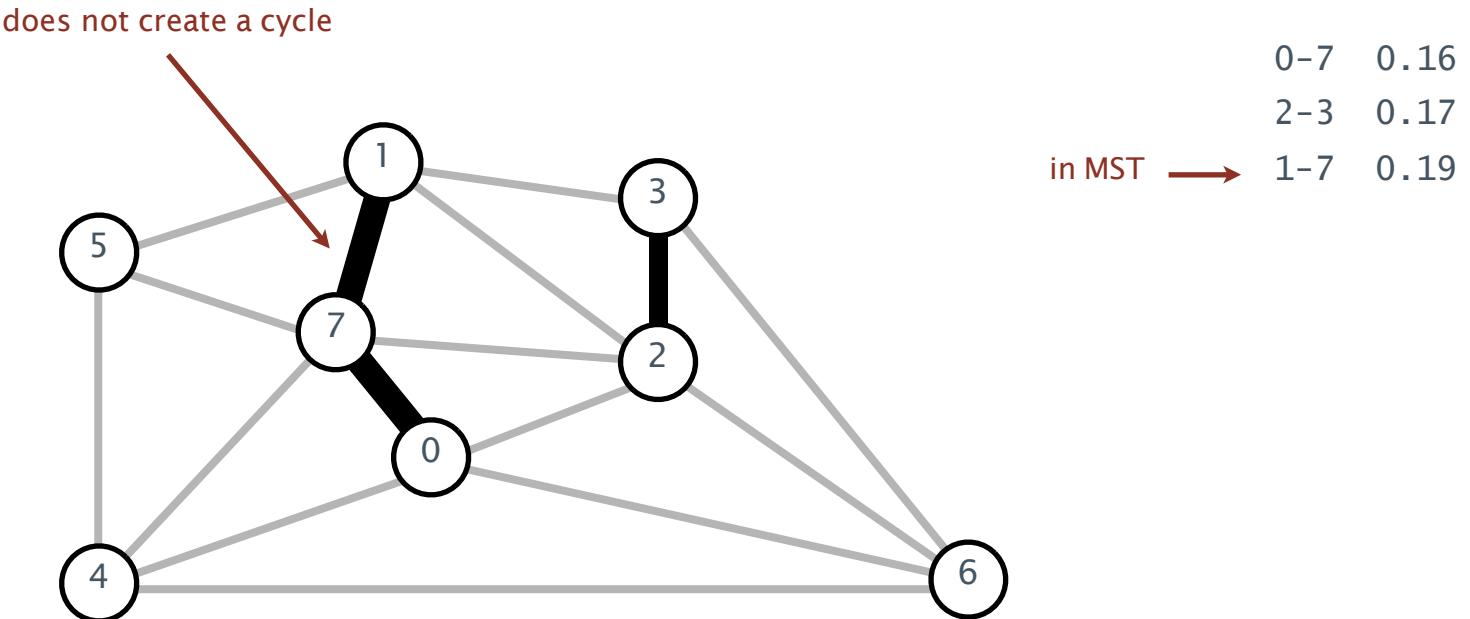
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

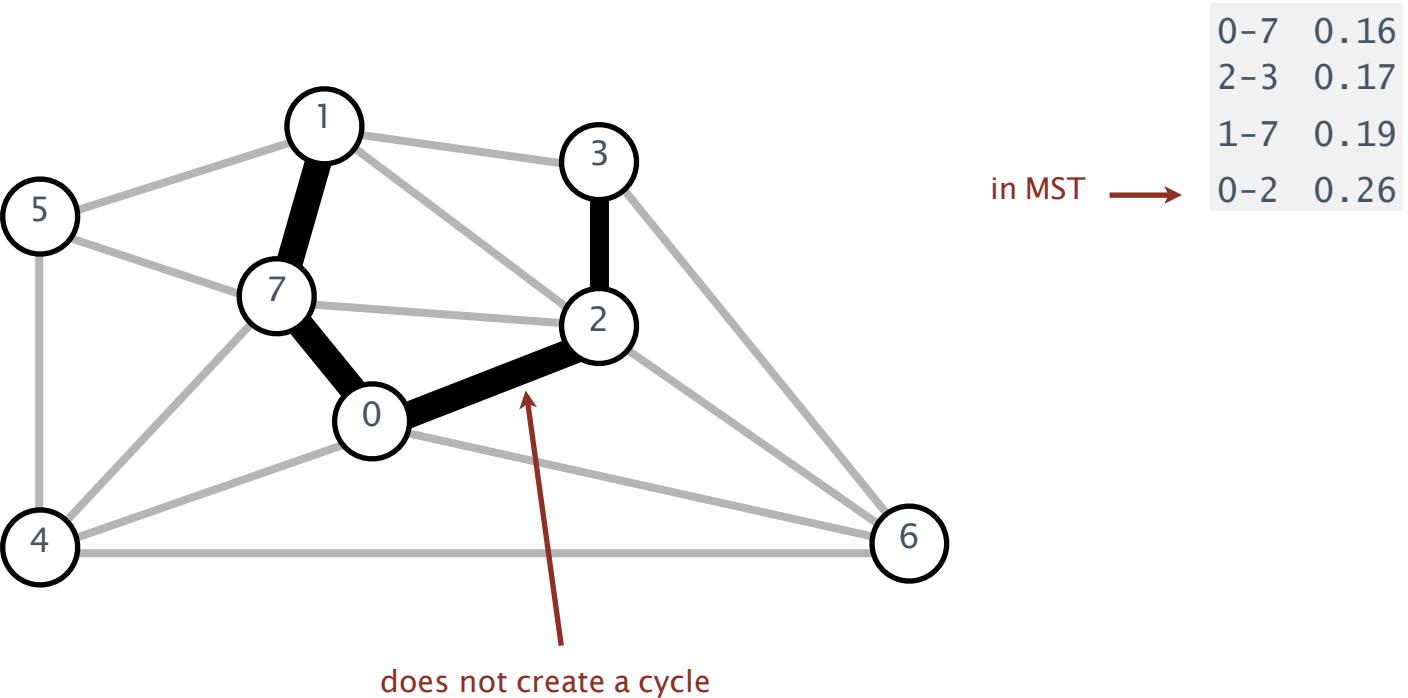
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

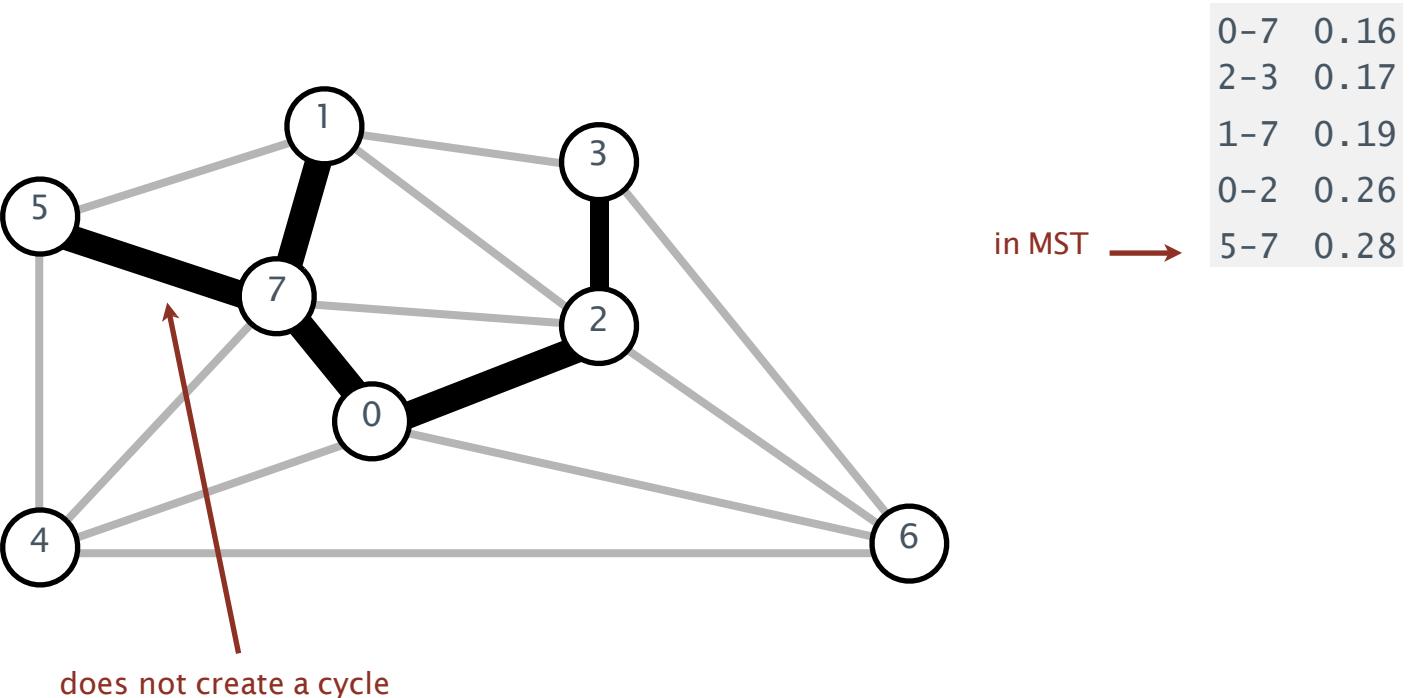
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

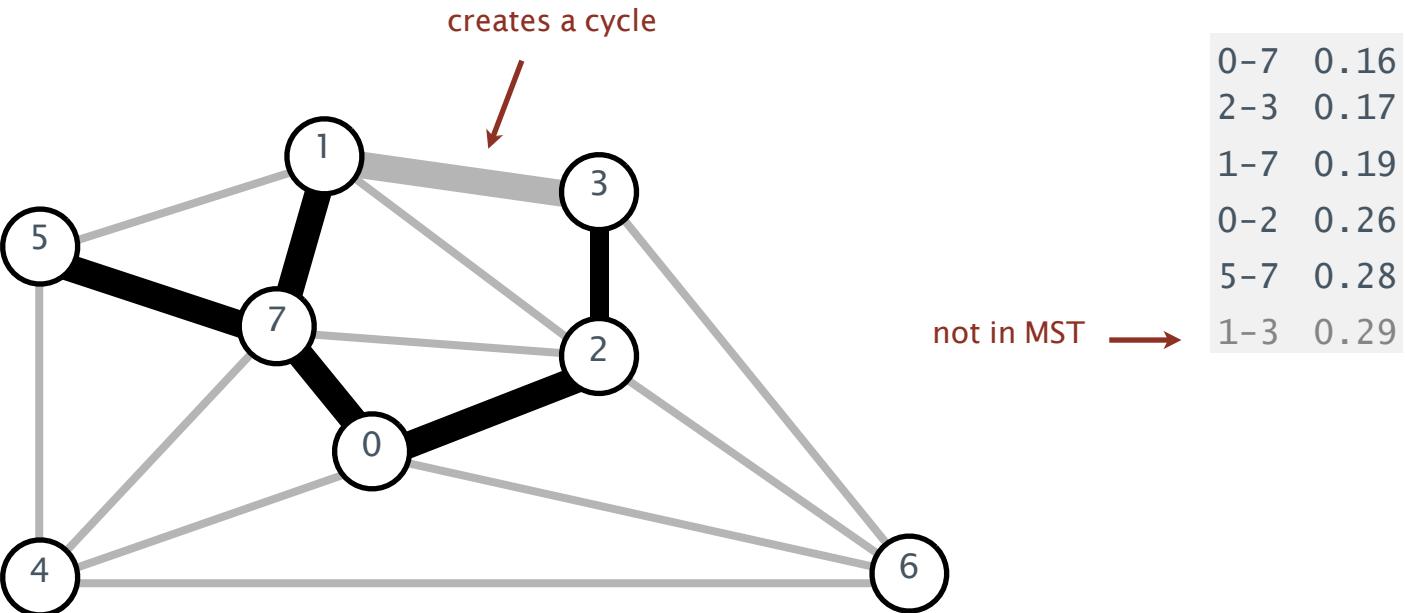
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

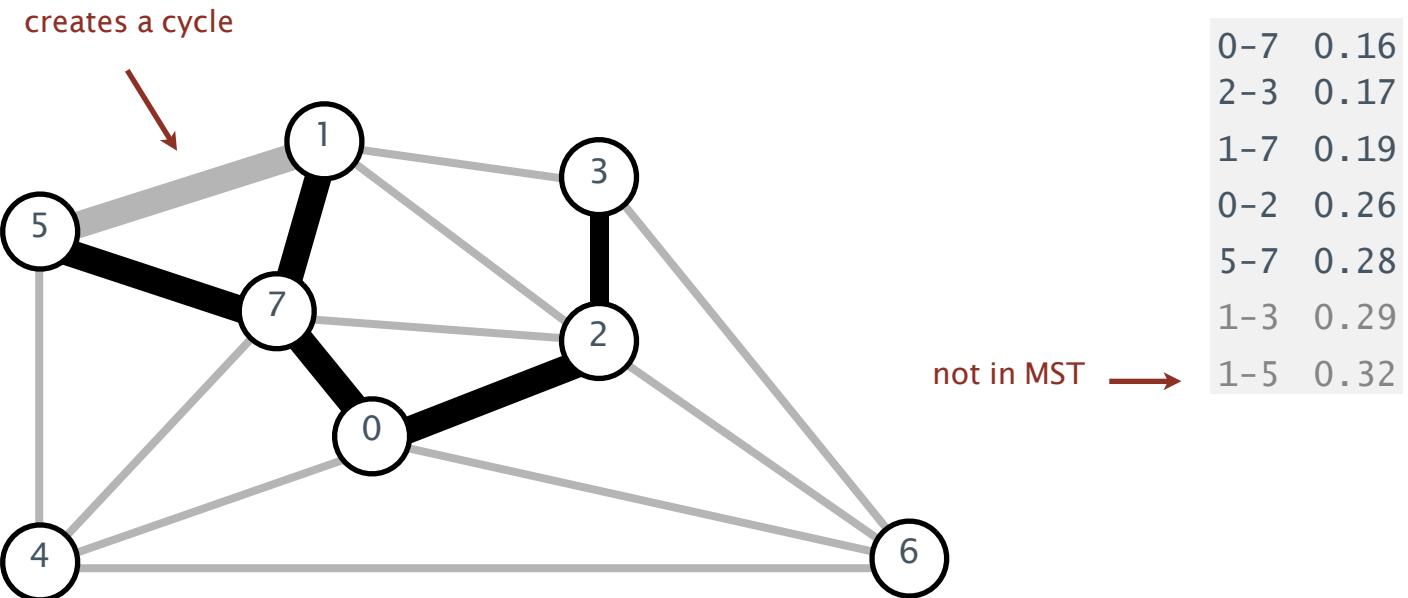
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

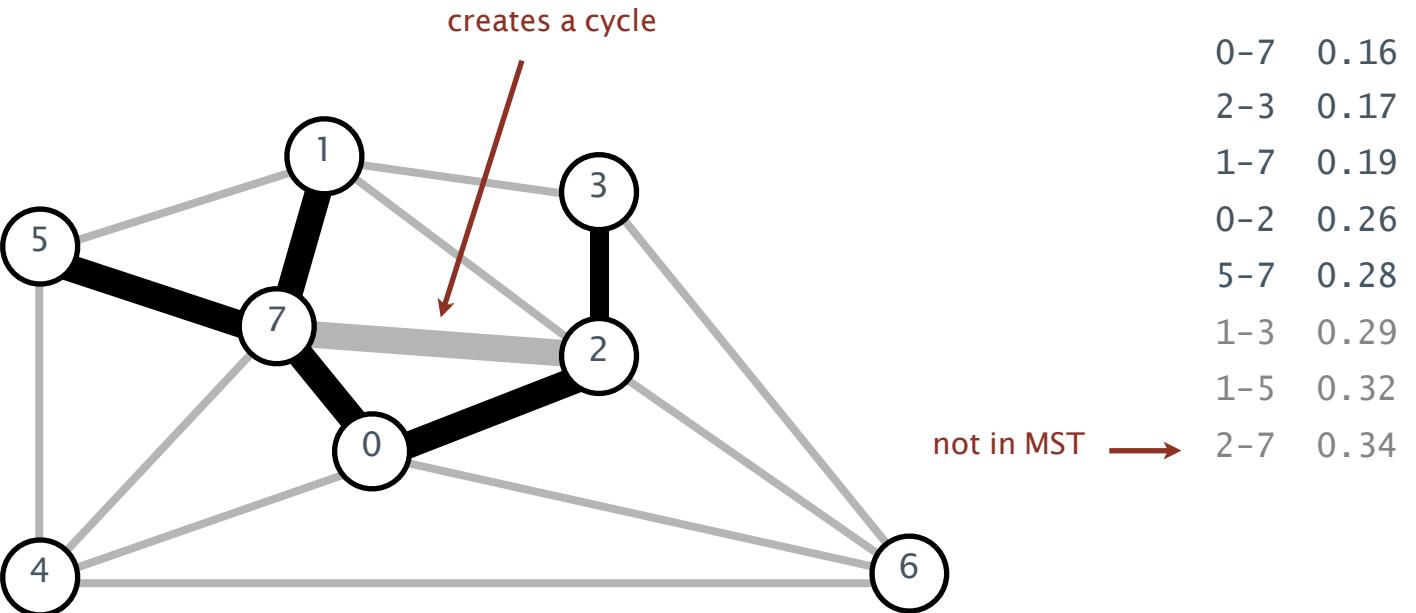
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

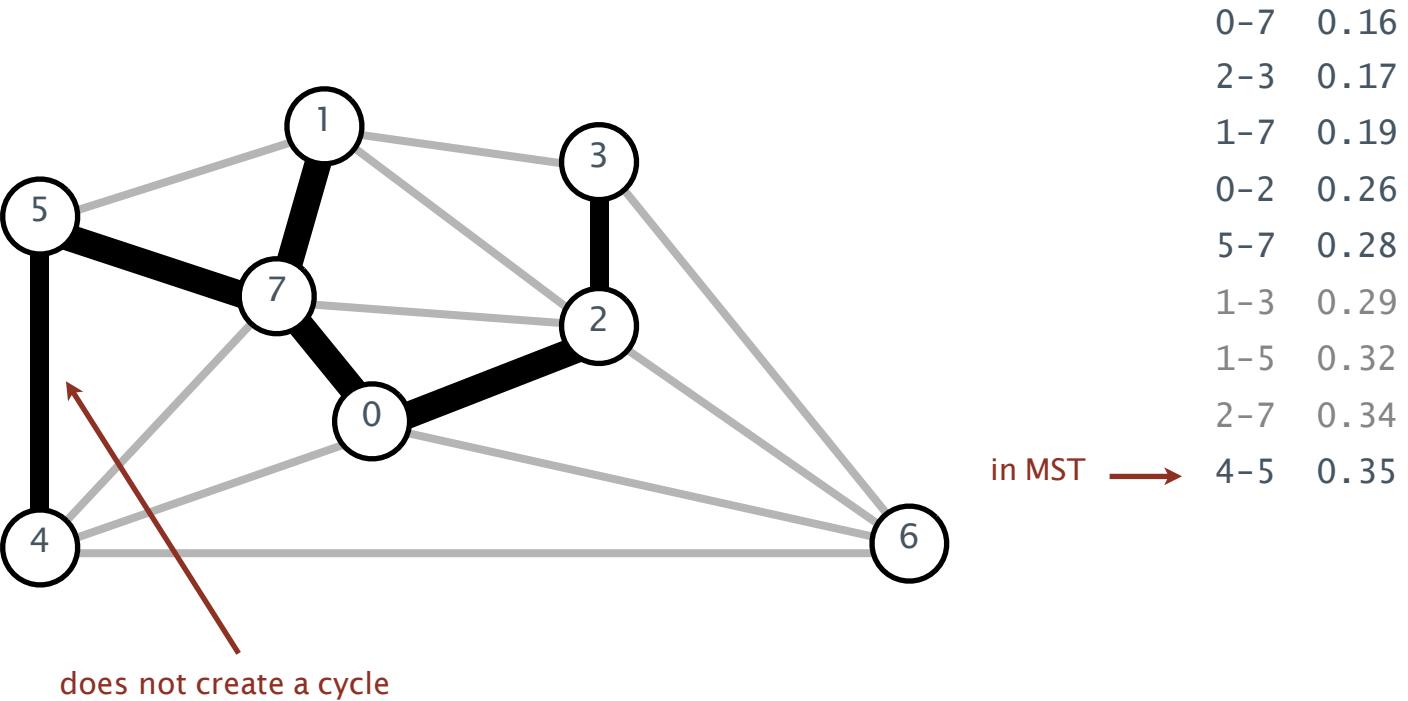
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

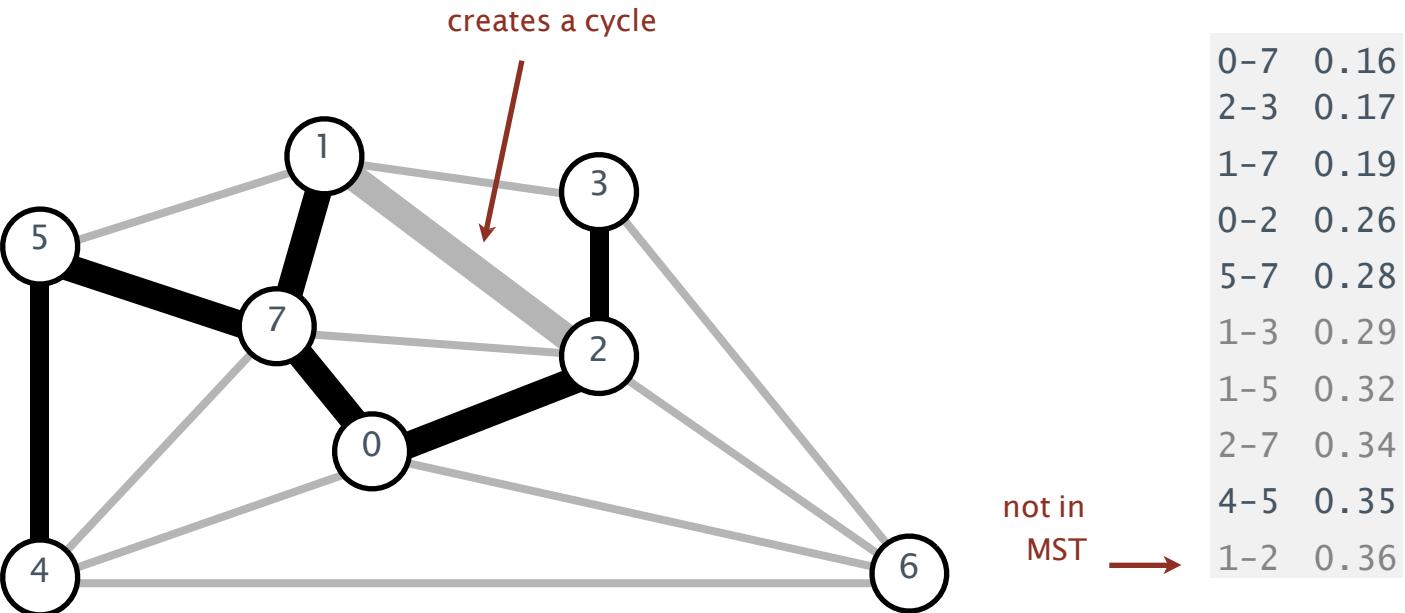
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

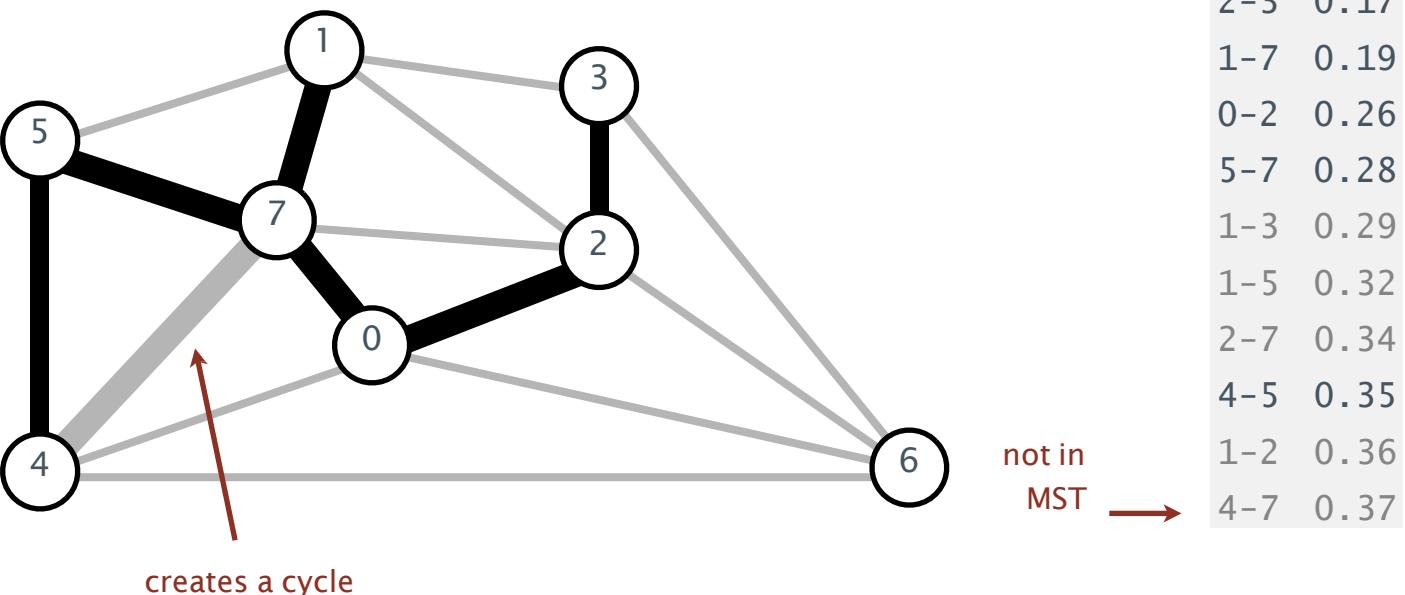
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

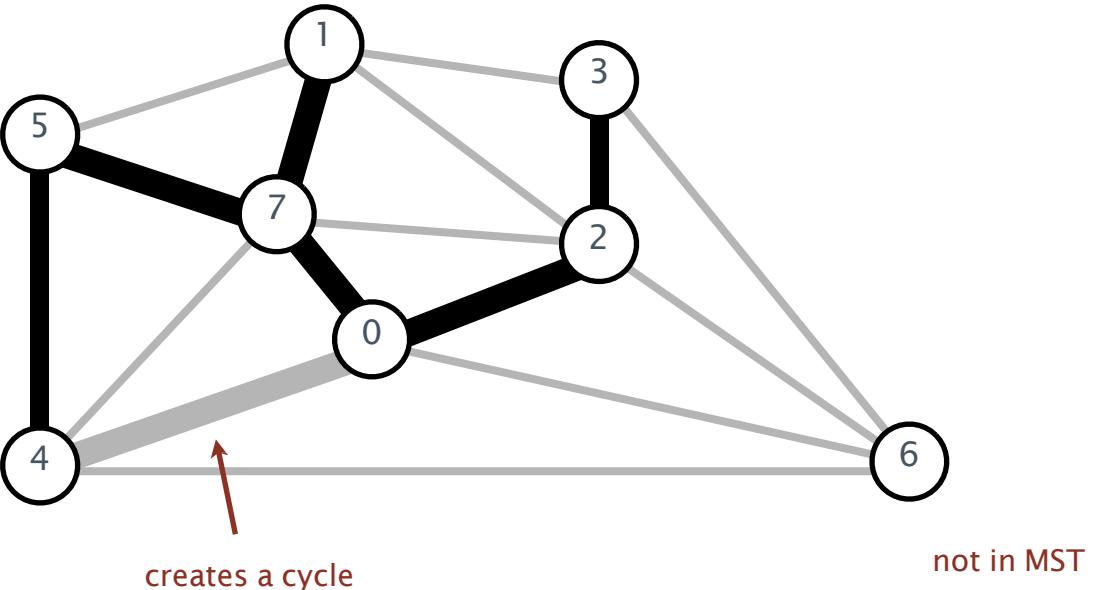
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree T unless doing so would create a cycle.

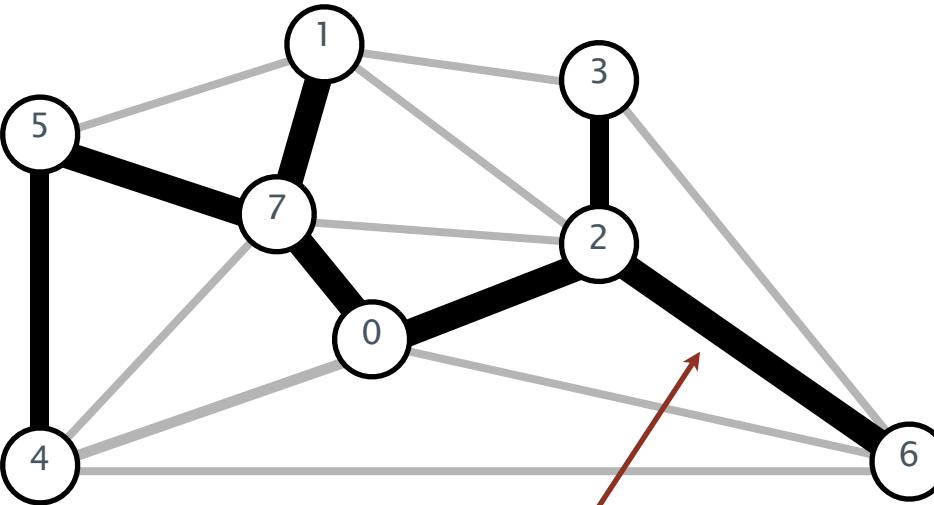


0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree T unless doing so would create a cycle.



does not create a cycle

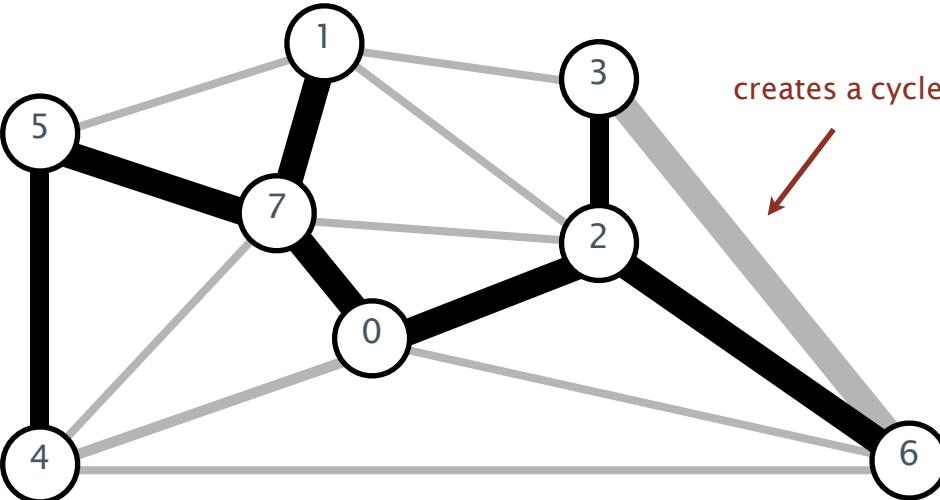
in MST →

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree T unless doing so would create a cycle.



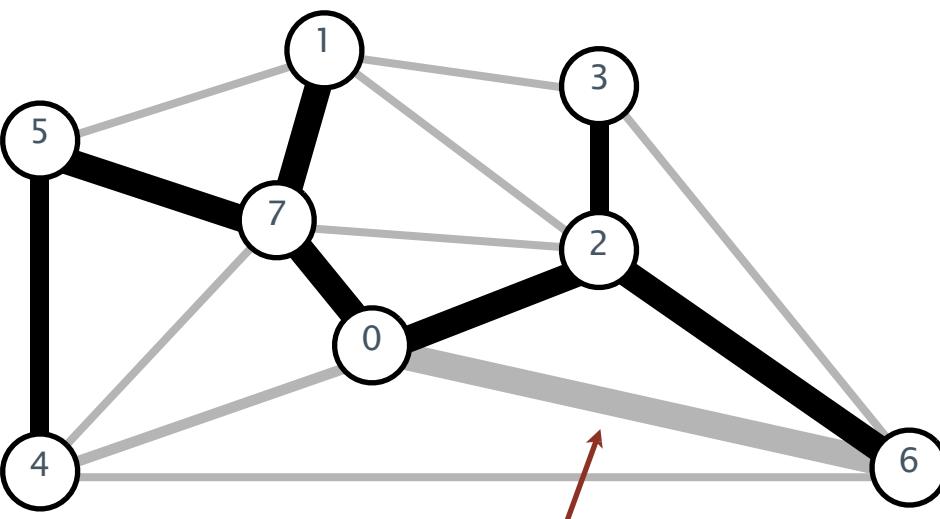
not in MST →

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree T unless doing so would create a cycle.



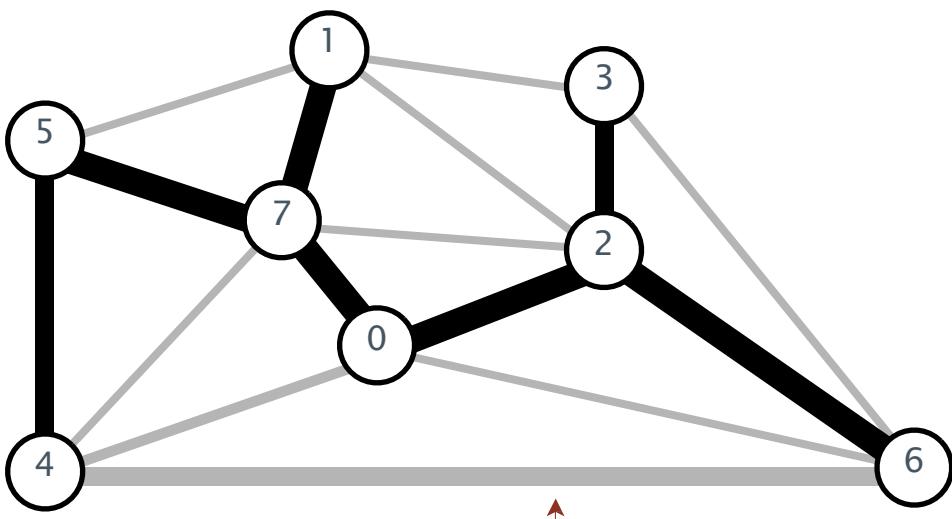
0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58

not in MST →

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree T unless doing so would create a cycle.



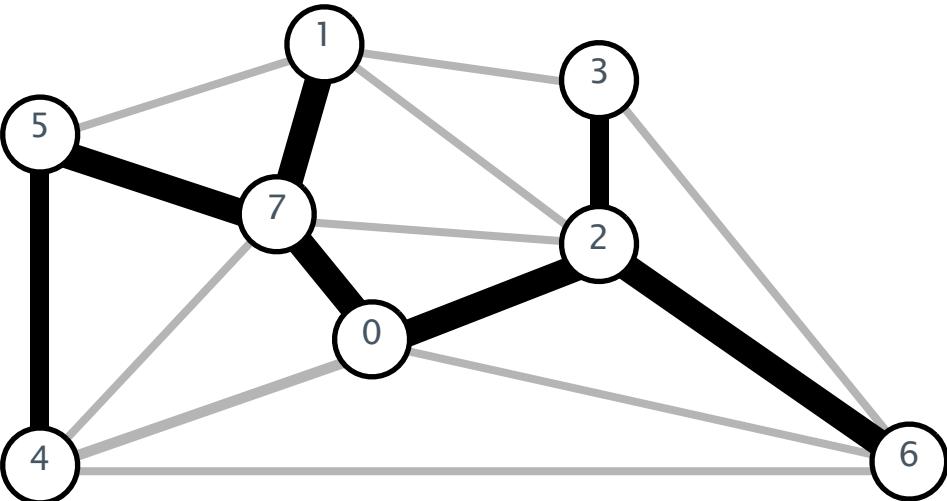
0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

not in MST →

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree T unless doing so would create a cycle.



a minimum spanning tree

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

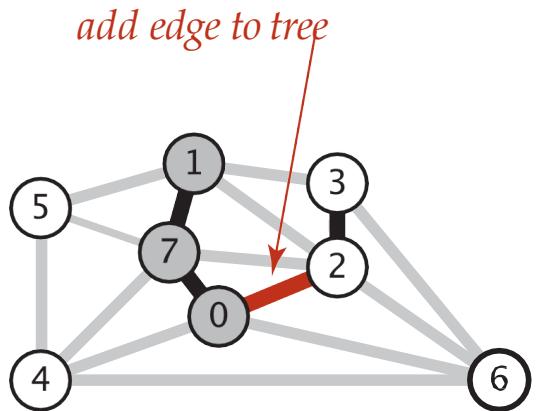
Kruskal's progression visualisation

Kruskal's algorithm: correctness proof

Proposition. [Kruskal 1956] Kruskal's algorithm computes the MST.

Pf. Kruskal's algorithm is a special case of the greedy MST algorithm.

- Suppose Kruskal's algorithm colors the edge $e = v-w$ black.
- Cut = set of vertices connected to v in tree T .
- No crossing edge is black.
- No crossing edge has lower weight

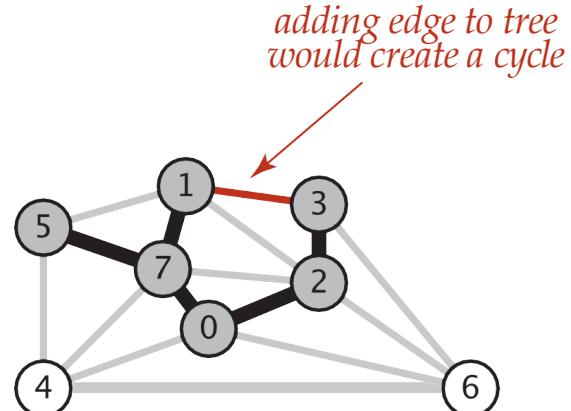
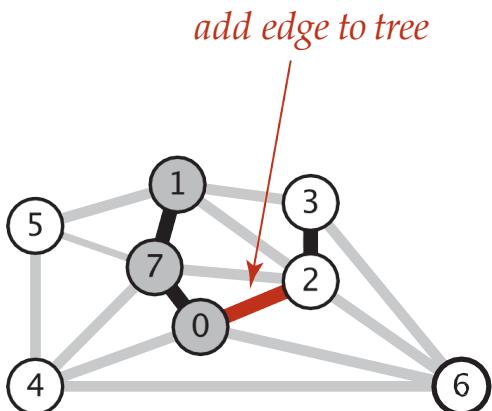


Kruskal's algorithm: implementation challenge

Challenge. Would adding edge $v-w$ to tree T create a cycle? If not, add it.

How difficult?

- $E + V$
- V ← run DFS from v , check if w is reachable
(T has at most $V - 1$ edges)
- $\log V$
- $\log^* V$ ← use the union-find data structure !
- 1

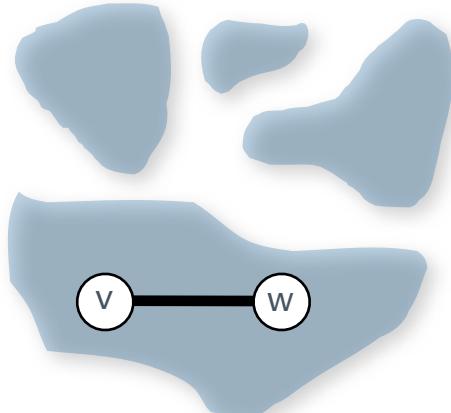


Kruskal's algorithm: implementation challenge

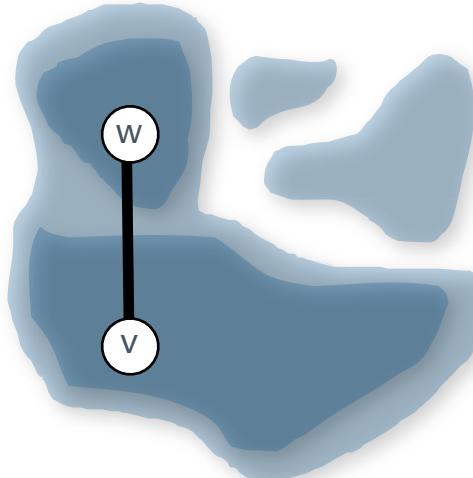
Challenge. Would adding edge $v-w$ to tree T create a cycle? If not, add it.

Efficient solution. Use the **union-find** data structure.

- Maintain a set for each connected component in T .
- If v and w are in same set, then adding $v-w$ would create a cycle.
- To add $v-w$ to T , merge sets containing v and w .



Case 1: adding $v-w$ creates a cycle



Case 2: add $v-w$ to T and merge sets containing v and w

Kruskal implementation

- › So what other data structures do we need?
 - Maintain the list of edges, ordered by weight, removing the lowest-weight edge when we add it to MST
 - List of edges and their weights added to the MST, to represent the MST (we'll need to iterate through them, and sum up their weight – to provide API required by MST)

Kruskal's algorithm: Java implementation

```
public class KruskalMST
{
    private Queue<Edge> mst = new Queue<Edge>();

    public KruskalMST(EdgeWeightedGraph G)
    {
        MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());

        UF uf = new UF(G.V());
        while (!pq.isEmpty() && mst.size() < G.V()-1)
        {
            Edge e = pq.delMin();
            int v = e.either(), w = e.other(v);
            if (!uf.connected(v, w))
            {
                uf.union(v, w);
                mst.enqueue(e);
            }
        }
    }

    public Iterable<Edge> edges()
    {   return mst;  }
}
```

build priority queue
(or sort)

greedily add edges to MST

edge v-w does not create cycle

merge sets

add edge to MST

Kruskal's algorithm: running time

Proposition. Kruskal's algorithm computes MST in time proportional to $E \log E$ (in the worst case).

Pf.	operation	frequency	time per op
	build pq	1	E
	delete-min	E	$\log E$
	union	V	$\log^* V^\dagger$
	connected	E	$\log^* V^\dagger$

† amortized bound using weighted quick union with path compression

recall: $\log^* V \leq 5$ in this universe

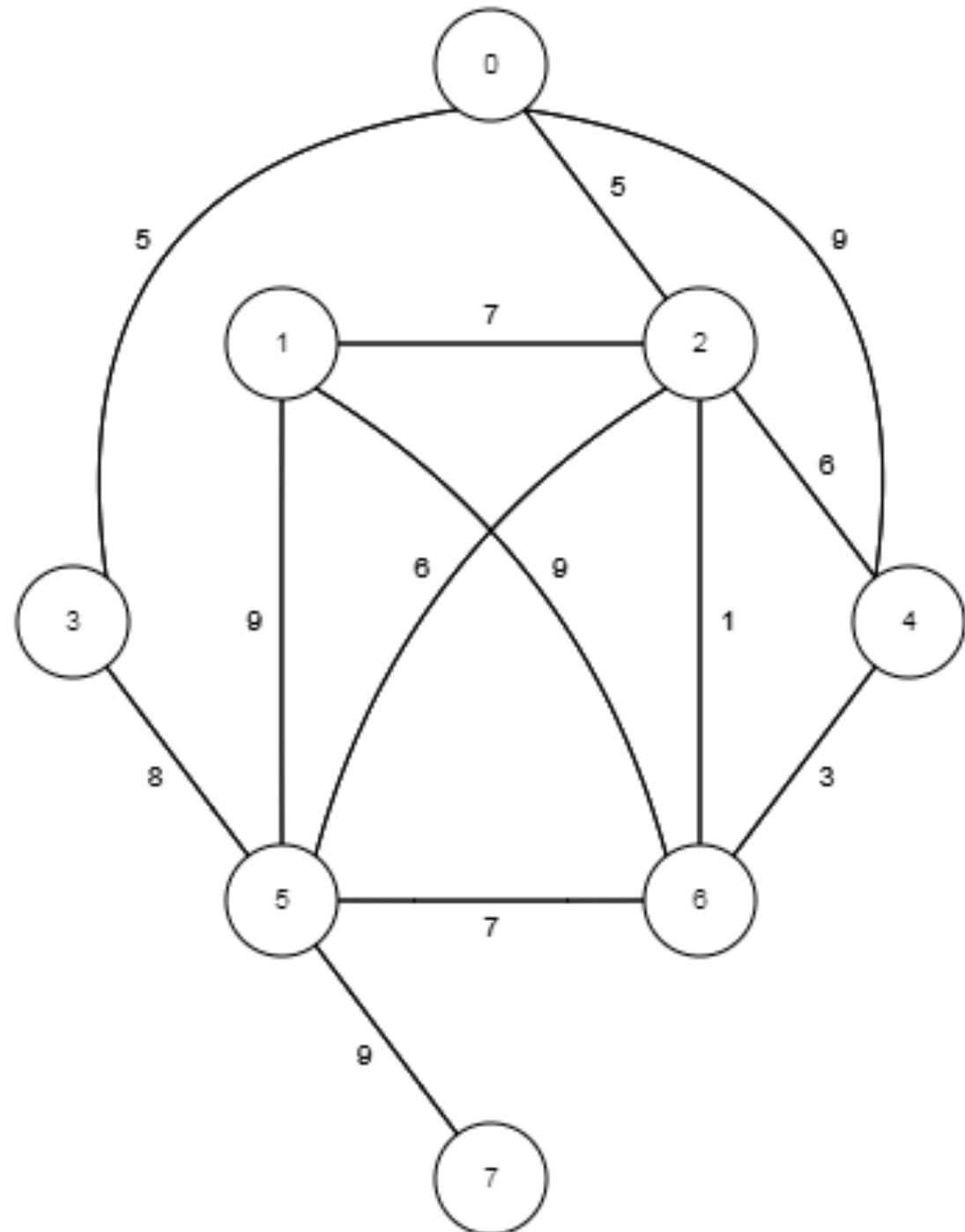


Remark. If edges are already sorted, order of growth is $E \log^* V$.

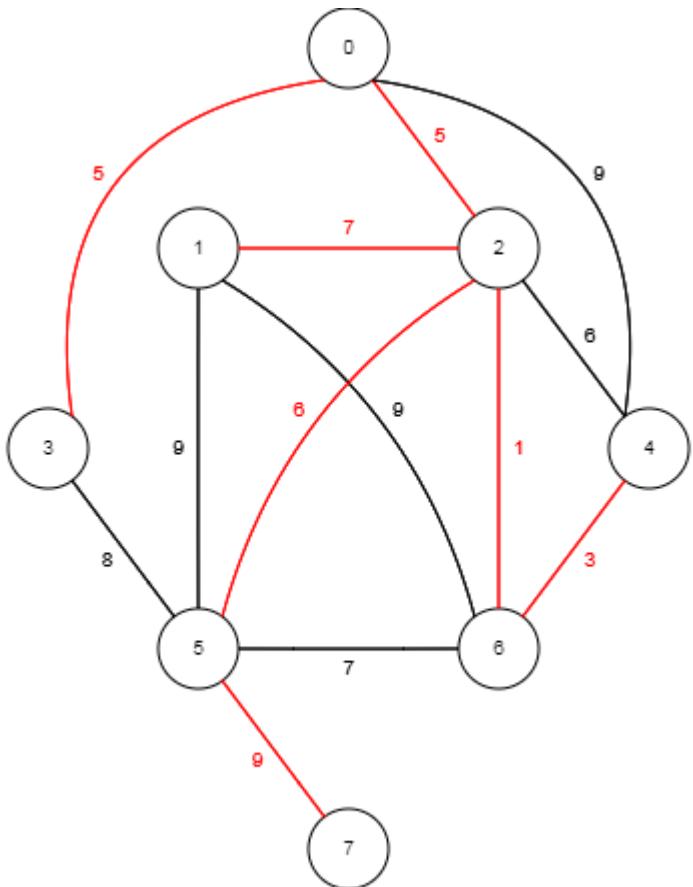
Kruskal's exercise

- › Apply Kruskal's algorithm to find an MST of the following graph
- › Provide trace of order in which edges are considered and added/discard from being added to an MST

v	w	Weight	Added to MST?
2	6	1	yes



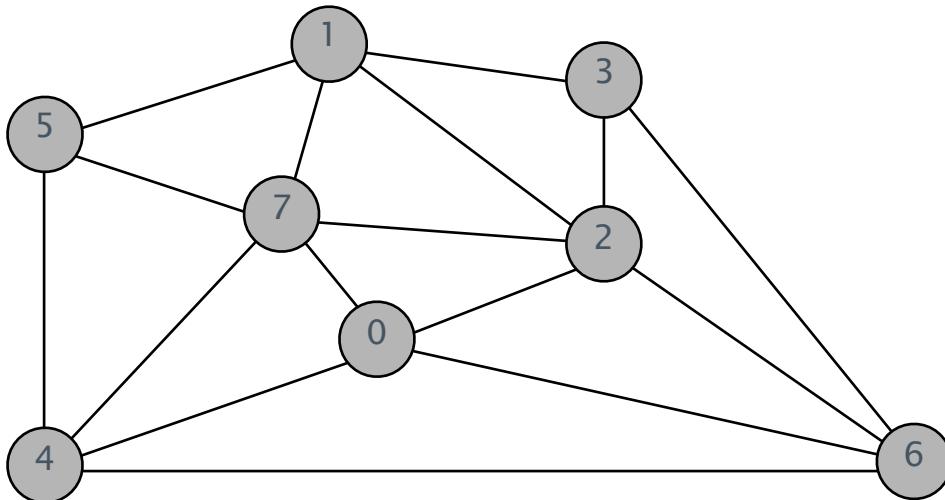
Kruskal's algorithm exercise solution



Prim's algorithm

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.

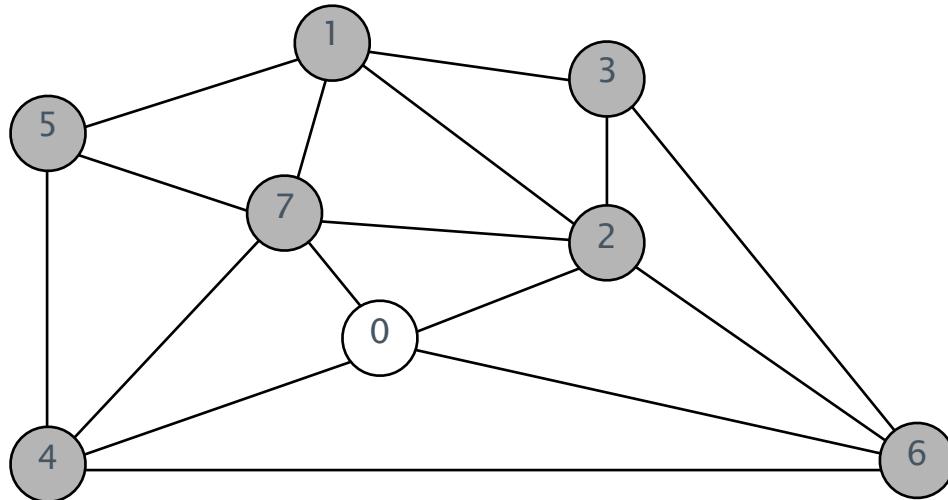


an edge-weighted graph

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

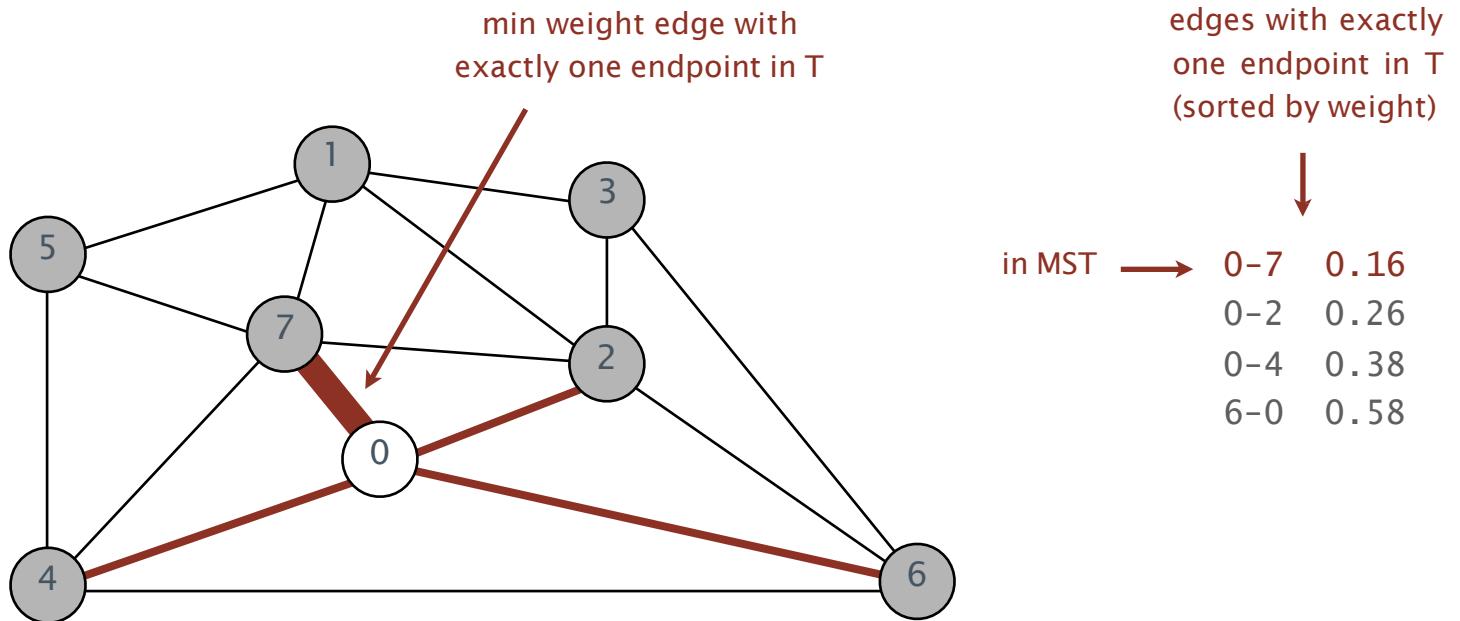
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



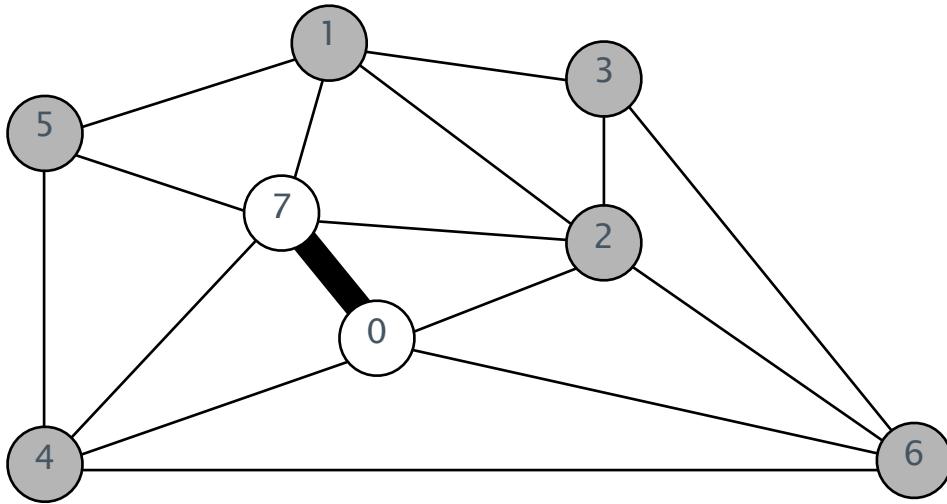
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.

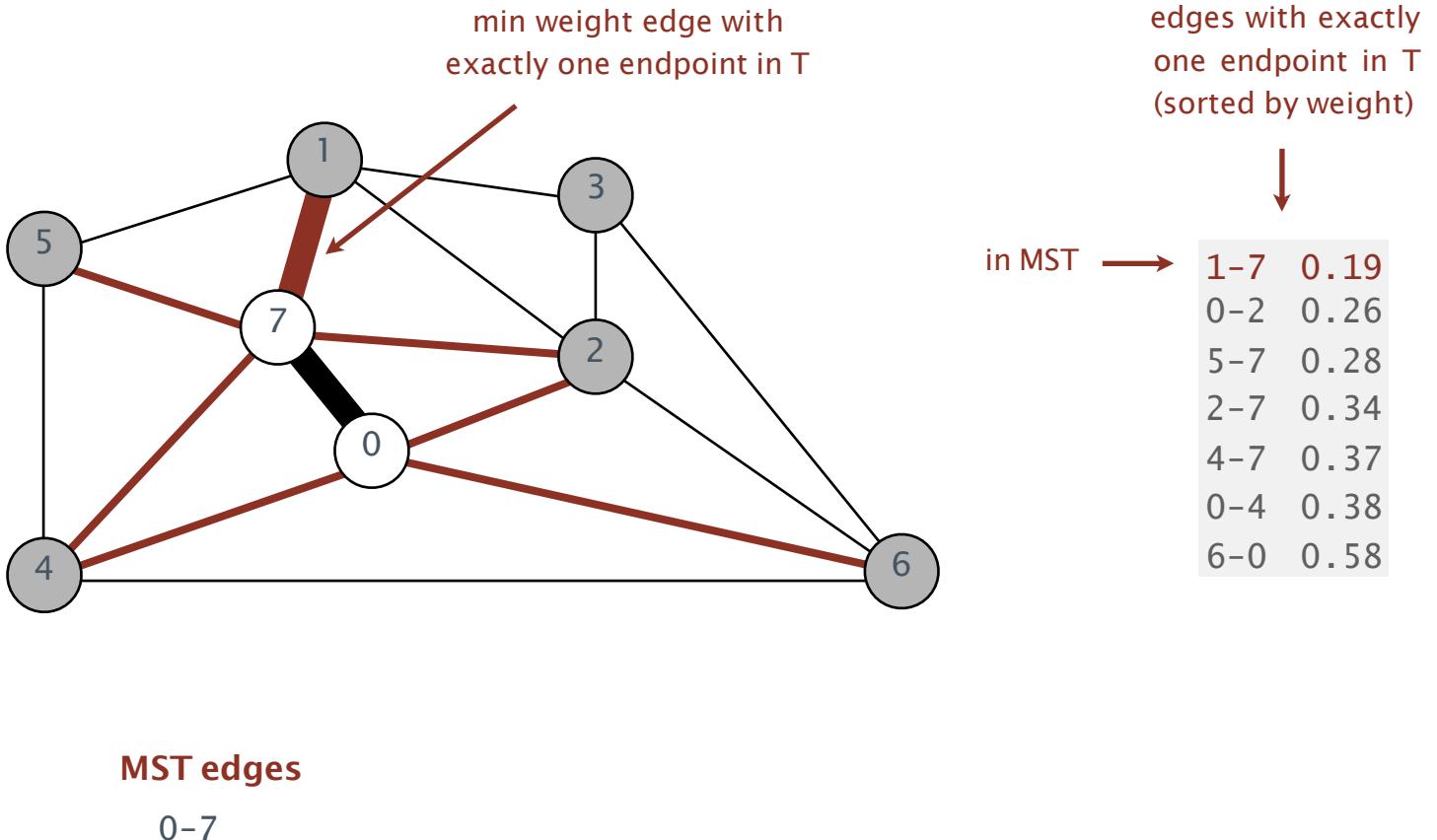


MST edges

0-7

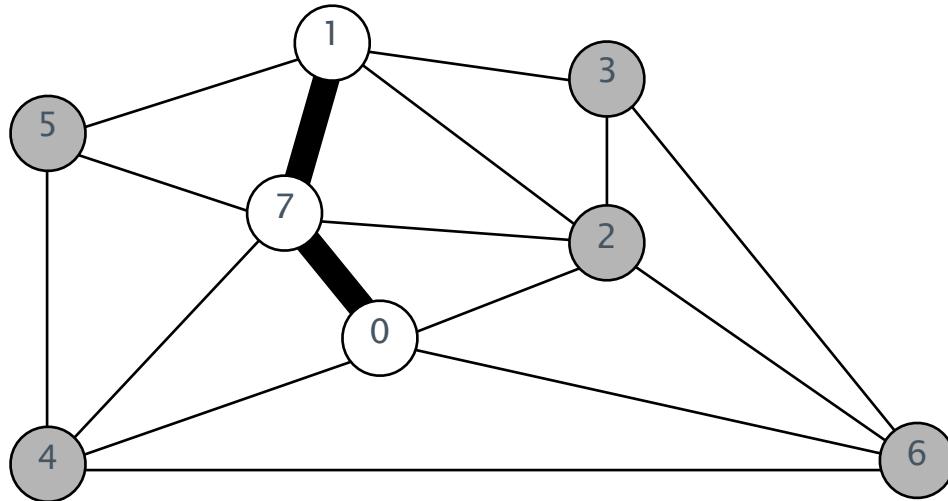
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.

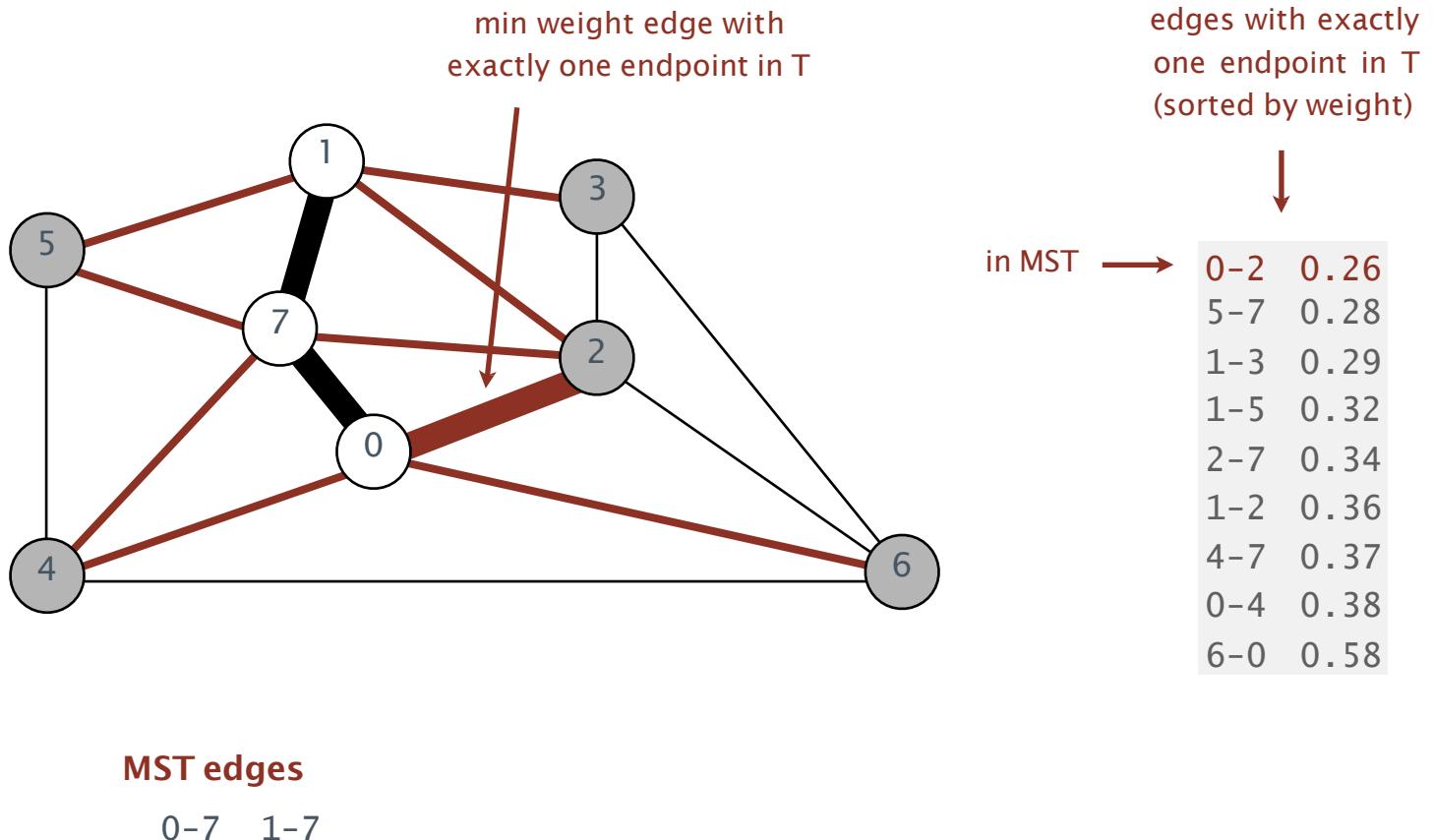


MST edges

0-7 1-7

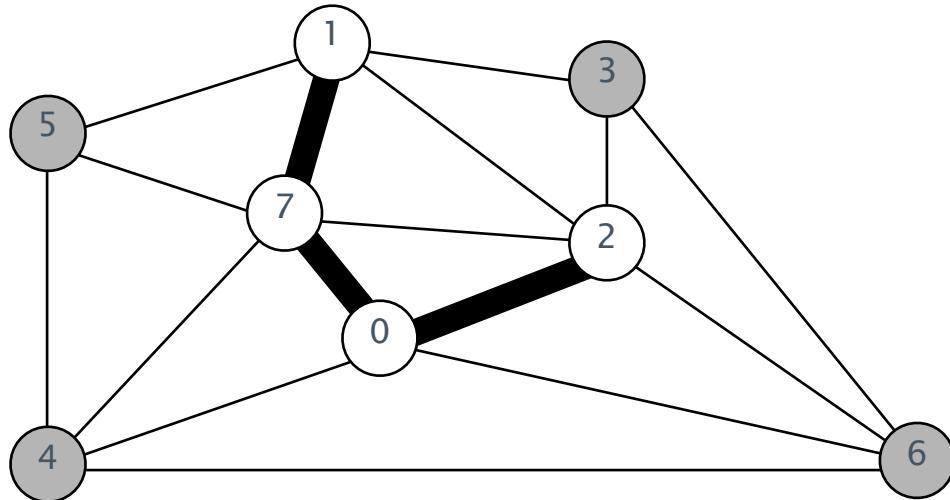
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.

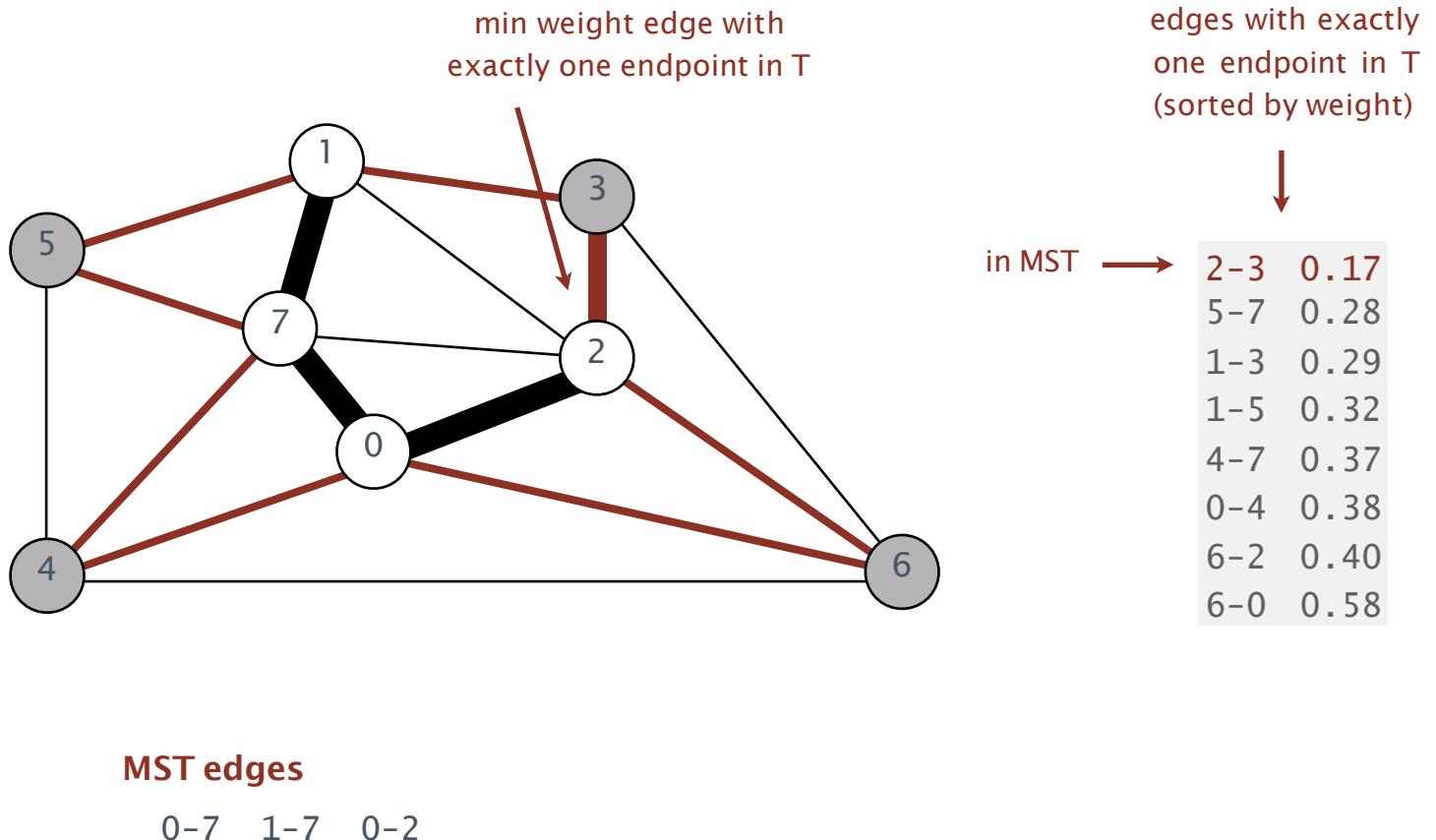


MST edges

0-7 1-7 0-2

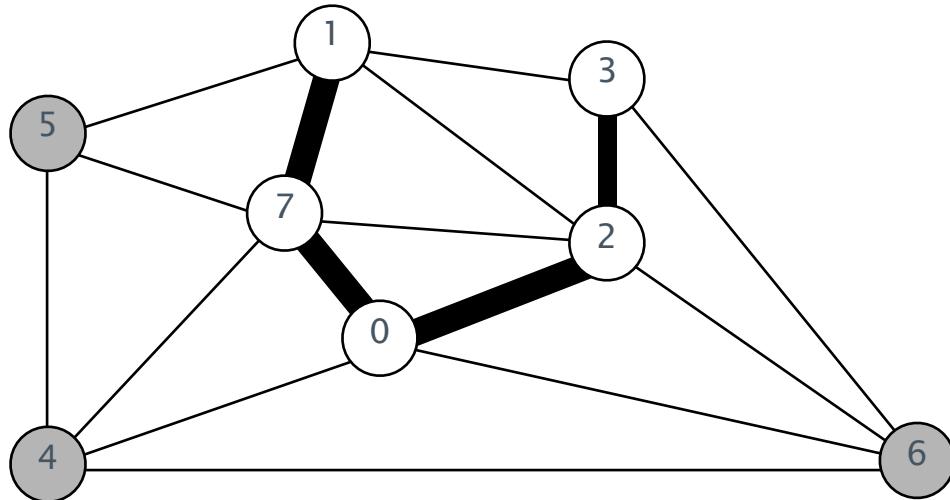
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.

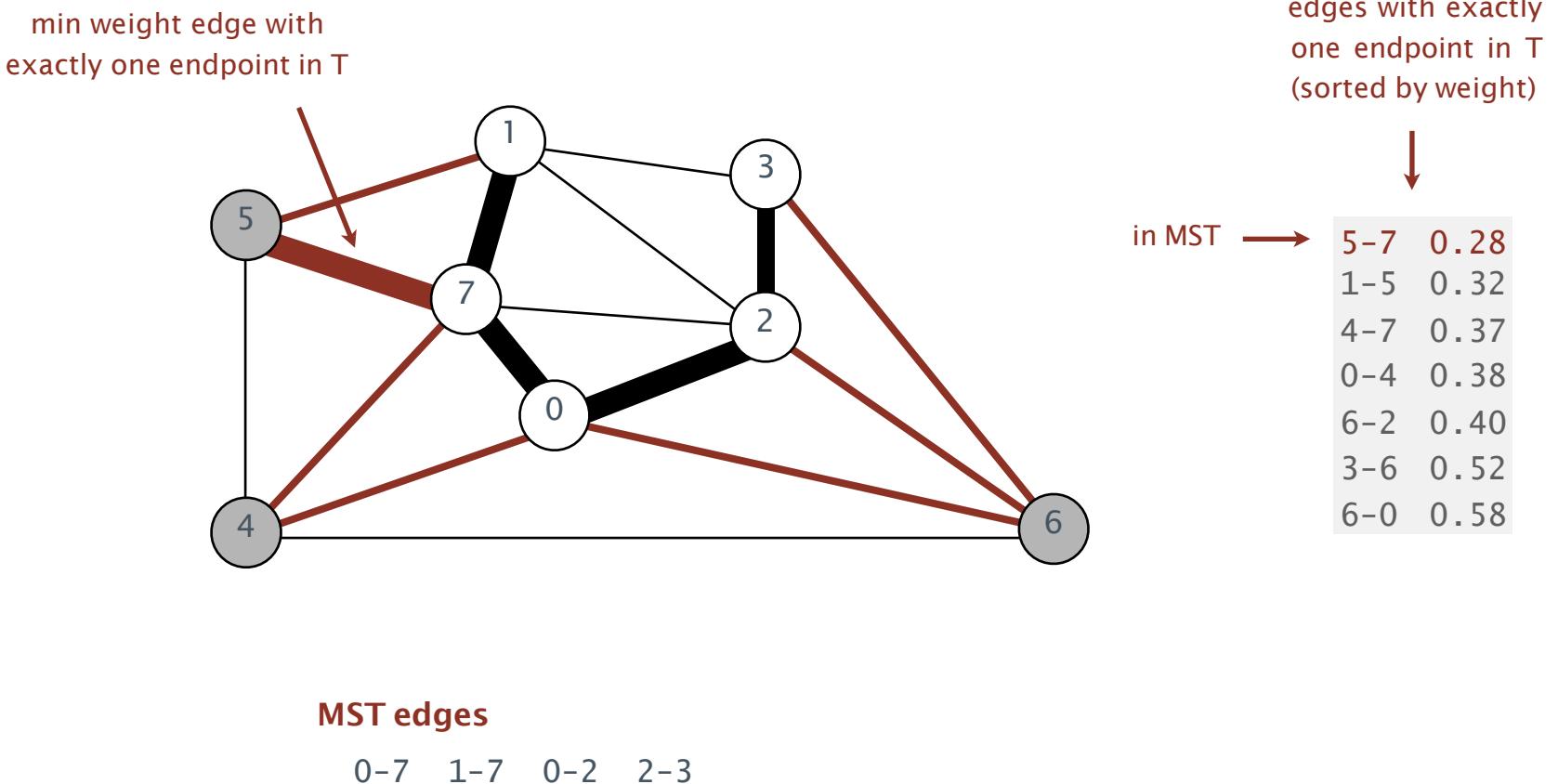


MST edges

0-7 1-7 0-2 2-3

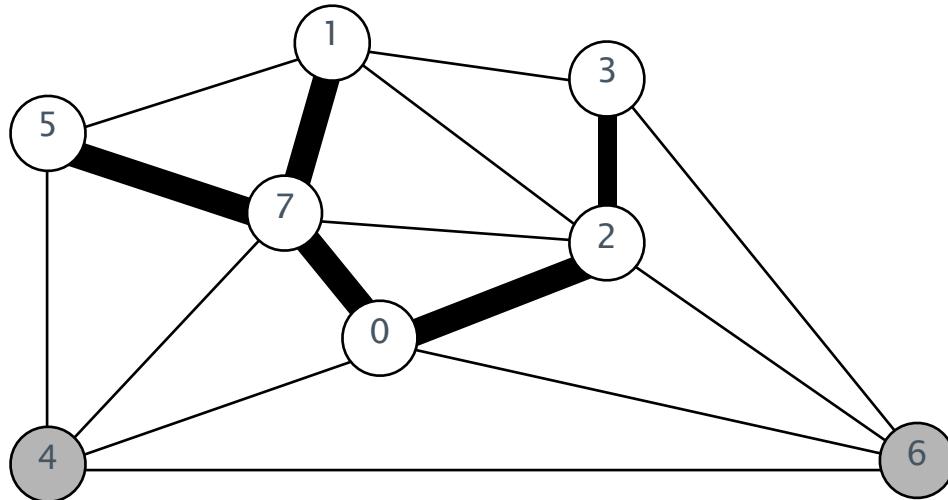
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.

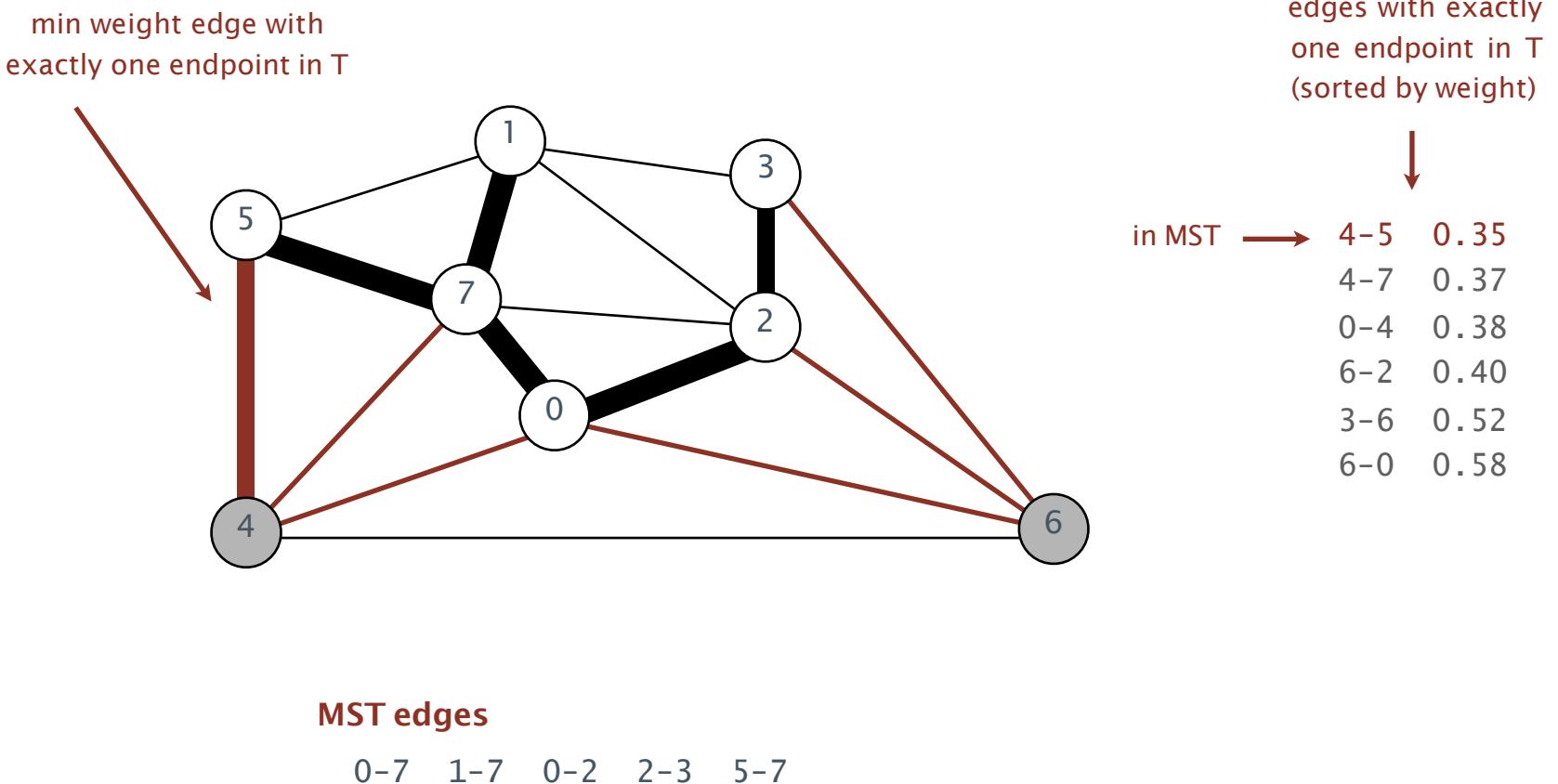


MST edges

0-7 1-7 0-2 2-3 5-7

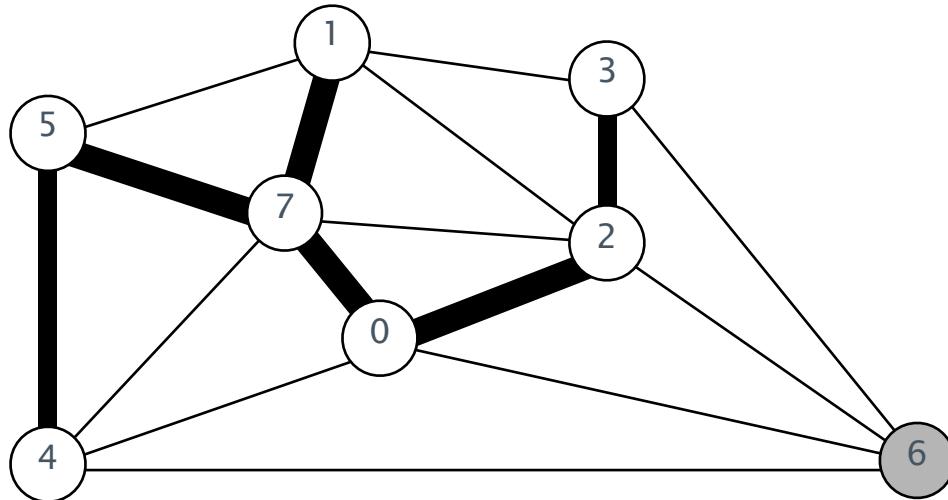
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.

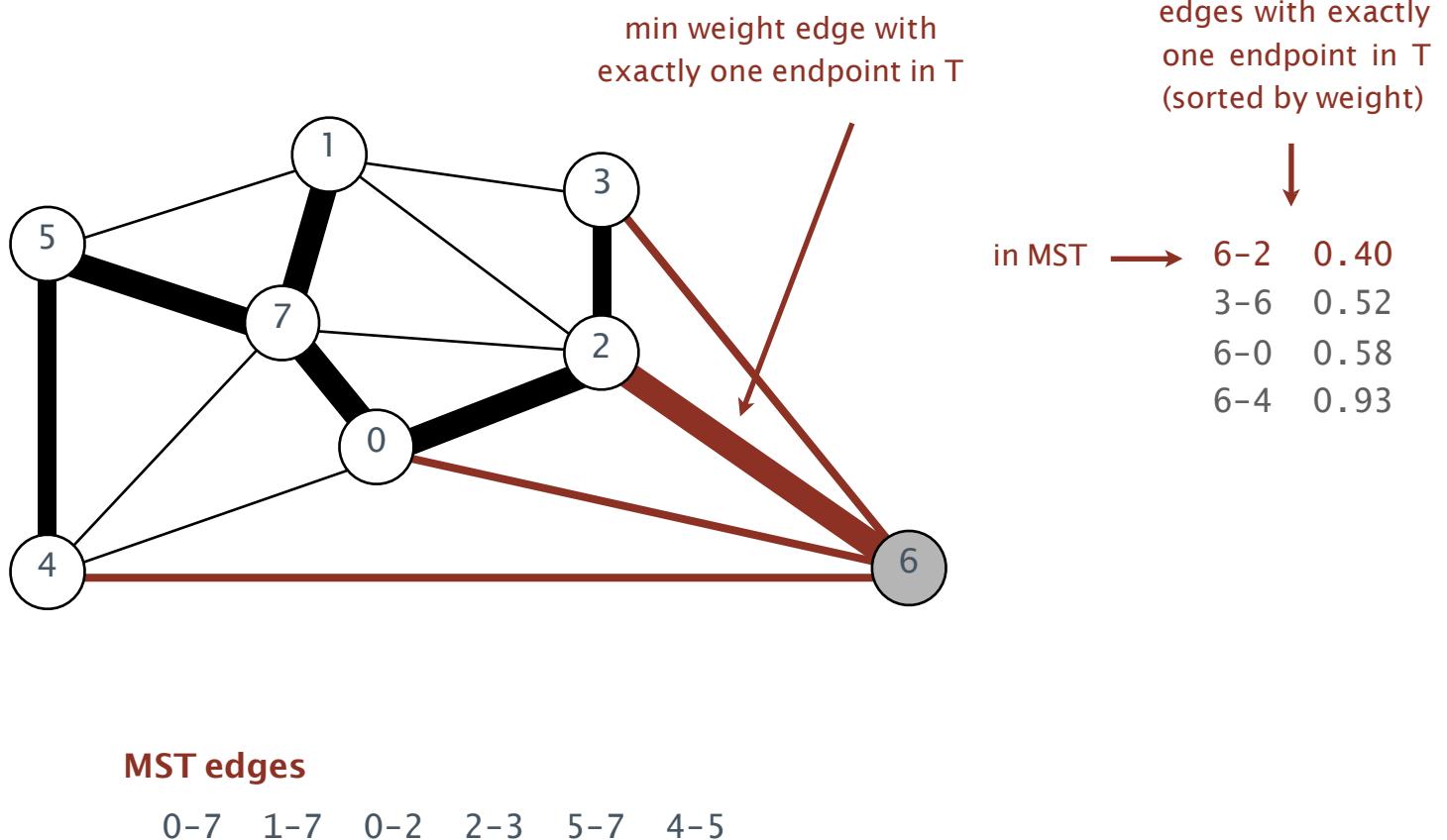


MST edges

0-7 1-7 0-2 2-3 5-7 4-5

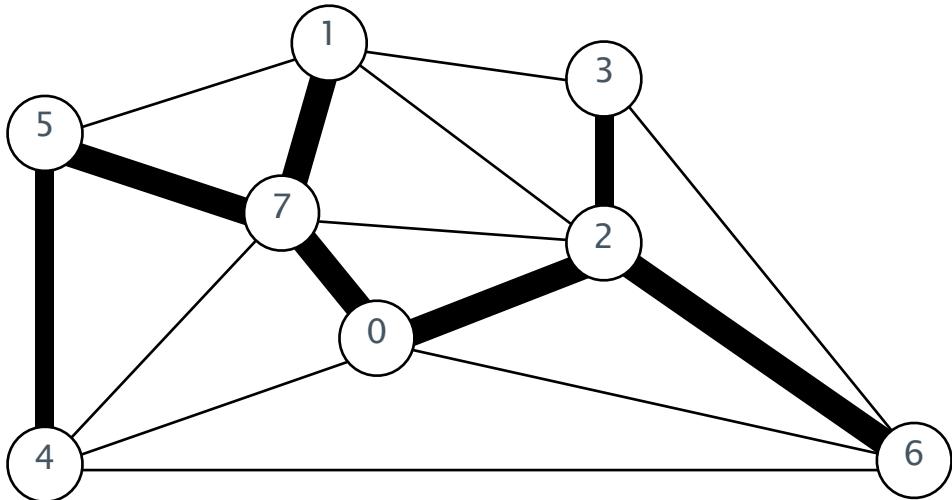
Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



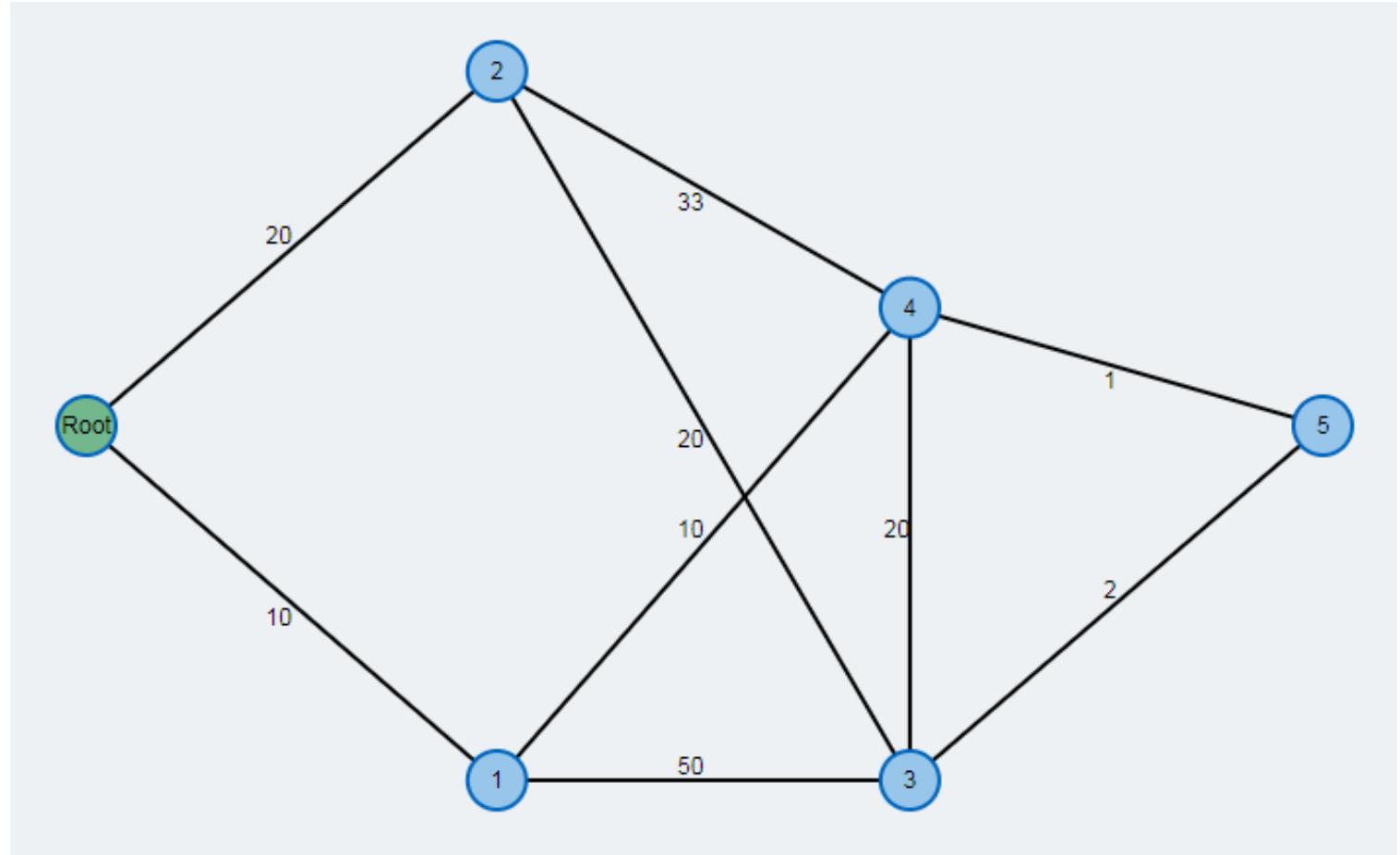
MST edges

0-7 1-7 0-2 2-3 5-7 4-5 6-2

Prim's algorithm visualisation

Prim's algorithm exercise- Turning Point

- › Starting from root (0)
- › show order in which edges are added to MST
- › Status of edgeTo and distance values



Prim implementation

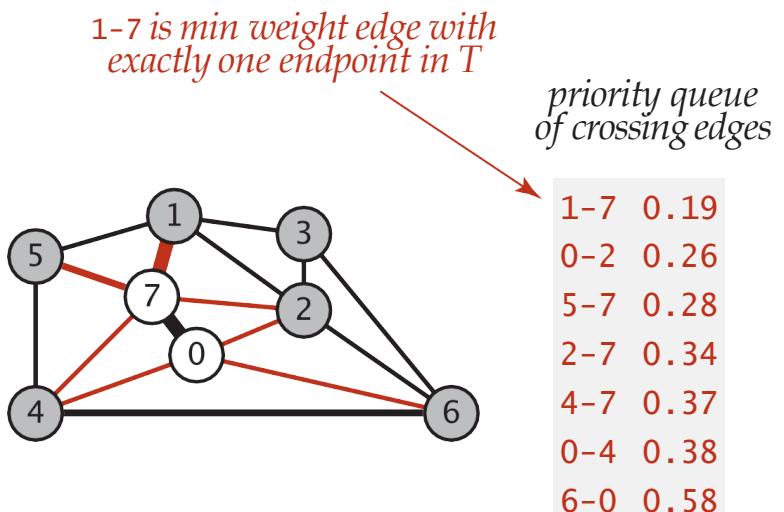
- › So what other data structures do we need?
 - Maintain the list of edges, ordered by weight, removing the lowest-weight edge when we add it to MST
 - › This list will be used slightly differently than in Kruskal – not list of all edges, just those with exactly one end-point in current MST
 - › Lazy vs Eager implementation
 - List of edges and their weights added to the MST, to represent the MST (we'll need to iterate through them, and sum up their weight – to provide API required by MST)

Prim's algorithm: lazy implementation

Challenge. Find the min weight edge with exactly one endpoint in T .

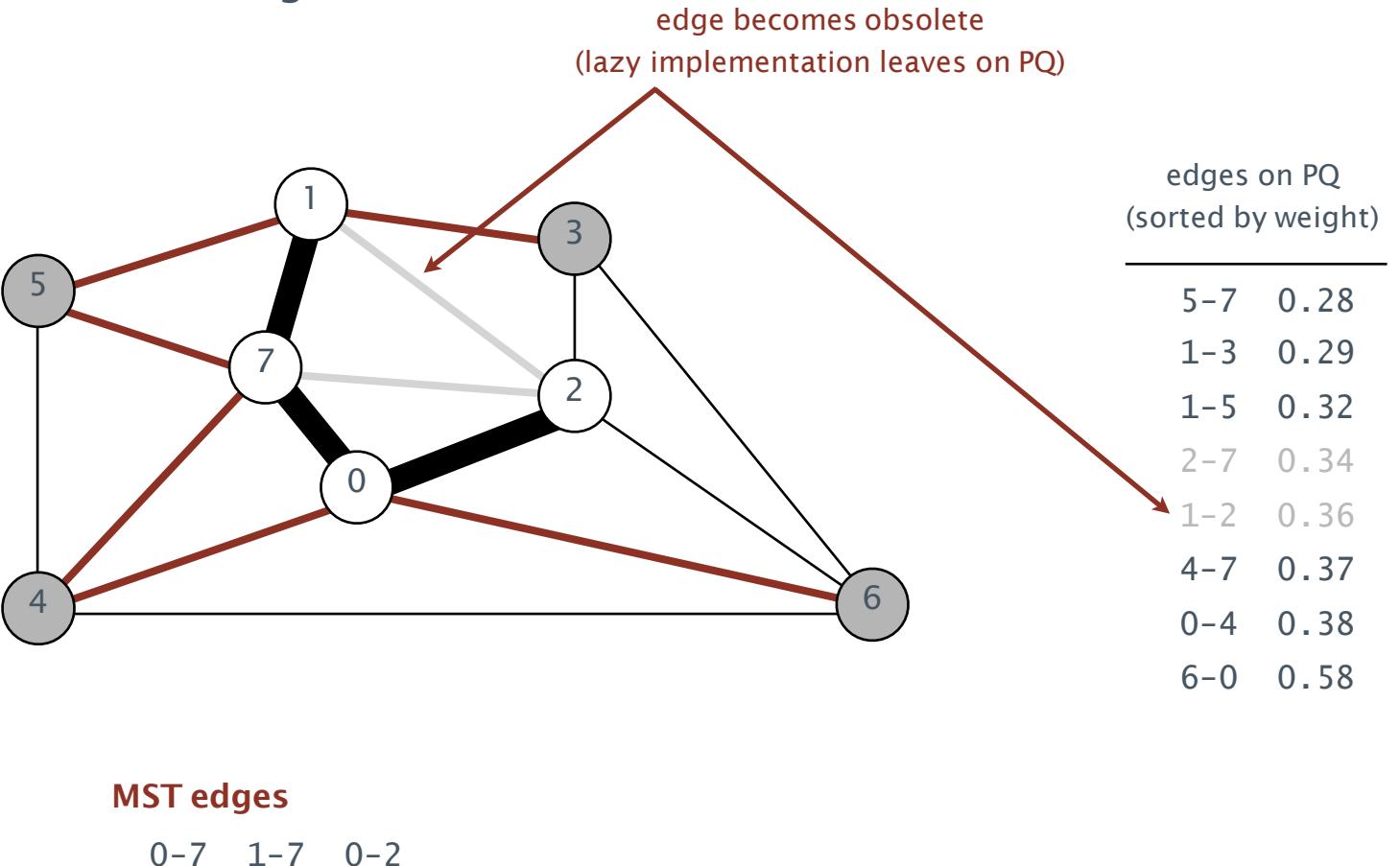
Lazy solution. Maintain a PQ of edges with (at least) one endpoint in T .

- Key = edge; priority = weight of edge.
- Delete-min to determine next edge $e = v-w$ to add to T .
- Disregard if both endpoints v and w are marked (both in T).
- Otherwise, let w be the unmarked vertex (not in T):
 - add to PQ any edge incident to w (assuming other endpoint not in T)
 - add e to T and mark w



Prim's algorithm: lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



Prim's algorithm: lazy implementation

```
public class LazyPrimMST
{
    private boolean[] marked; // MST
                           vertices
    private Queue<Edge> mst; // MST edges
    private MinPQ<Edge> pq; // PQ of edges
    public LazyPrimMST(WeightedGraph G)
    {
        pq = new MinPQ<Edge>();
        mst = new Queue<Edge>();
        marked = new boolean[G.V()];
        visit(G, 0);
    }

    while (!pq.isEmpty() && mst.size() < G.V() - 1)
    {
        Edge e = pq.delMin();
        int v = e.either(), w = e.other(v);
        if (marked[v] && marked[w]) continue;
        mst.enqueue(e);
        if (!marked[v]) visit(G, v);
        if (!marked[w]) visit(G, w);
    }
}
```

assume G is connected

repeatedly delete the min weight edge $e = v-w$ from PQ

ignore if both endpoints in T

add edge e to tree

add v or w to tree

Prim's algorithm: lazy implementation

```
private void visit(WeightedGraph G, int v)
{
    marked[v] = true;
    for (Edge e : G.adj(v))
        if (!marked[e.other(v)])
            pq.insert(e);
}

public Iterable<Edge> mst()
{   return mst; }
```

add v to T

for each edge $e = v-w$, add to PQ if w not already in T

Lazy Prim's algorithm: running time

Proposition. Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to E (in the worst case).

Pf.

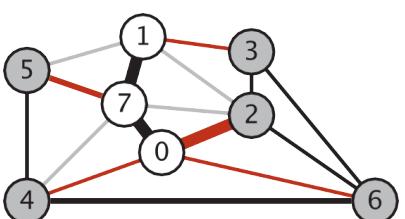
	operation	frequency	binary heap
	delete min	E	$\log E$
	insert	E	$\log E$

Prim's algorithm: eager implementation

Challenge. Find min weight edge with exactly one endpoint in T .

Eager solution. Maintain a PQ of vertices connected by an edge to T , where priority of vertex v = weight of shortest edge connecting v to T .

- Delete min vertex v and add its associated edge $e = v-w$ to T .
- Update PQ by considering all edges $e = v-x$ incident to v
 - ignore if x is already in T
 - add x to PQ if not already on it
 - decrease priority of x if $v-x$ becomes shortest edge connecting x to T



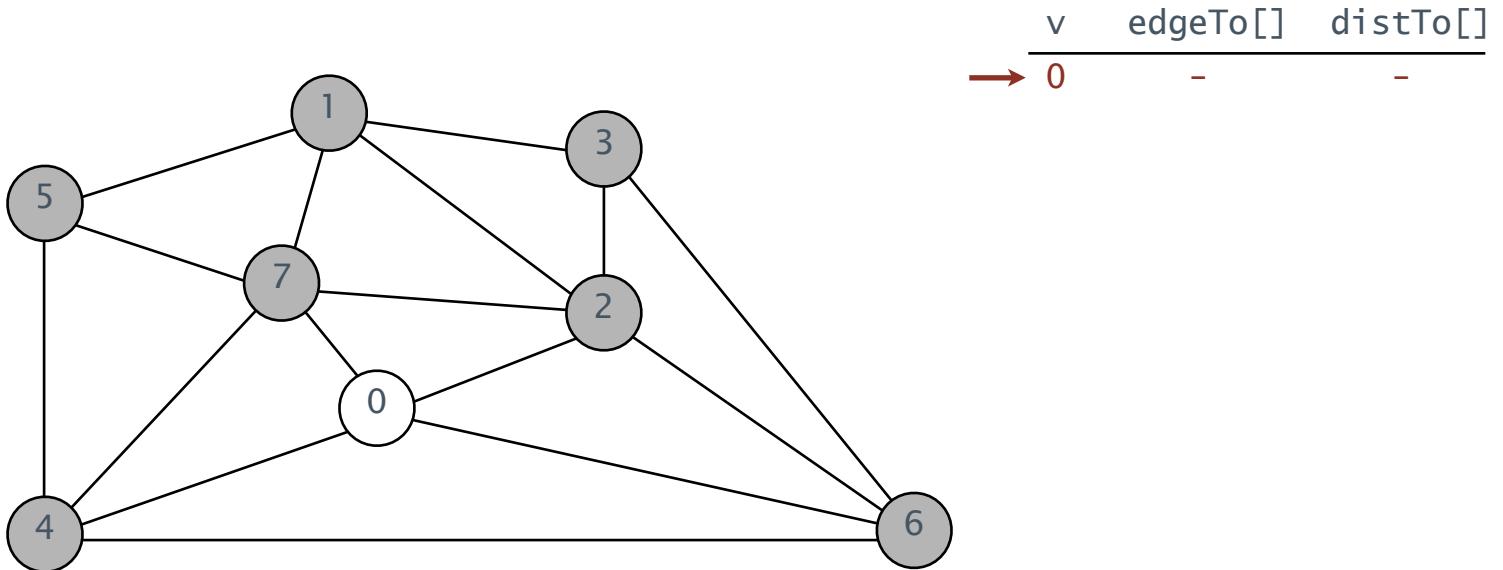
0		
1	1-7	0.19
2	0-2	0.26
3	1-3	0.29
4	0-4	0.38
5	5-7	0.28
6	6-0	0.58
7	0-7	0.16

red: onPQ

black: onMST

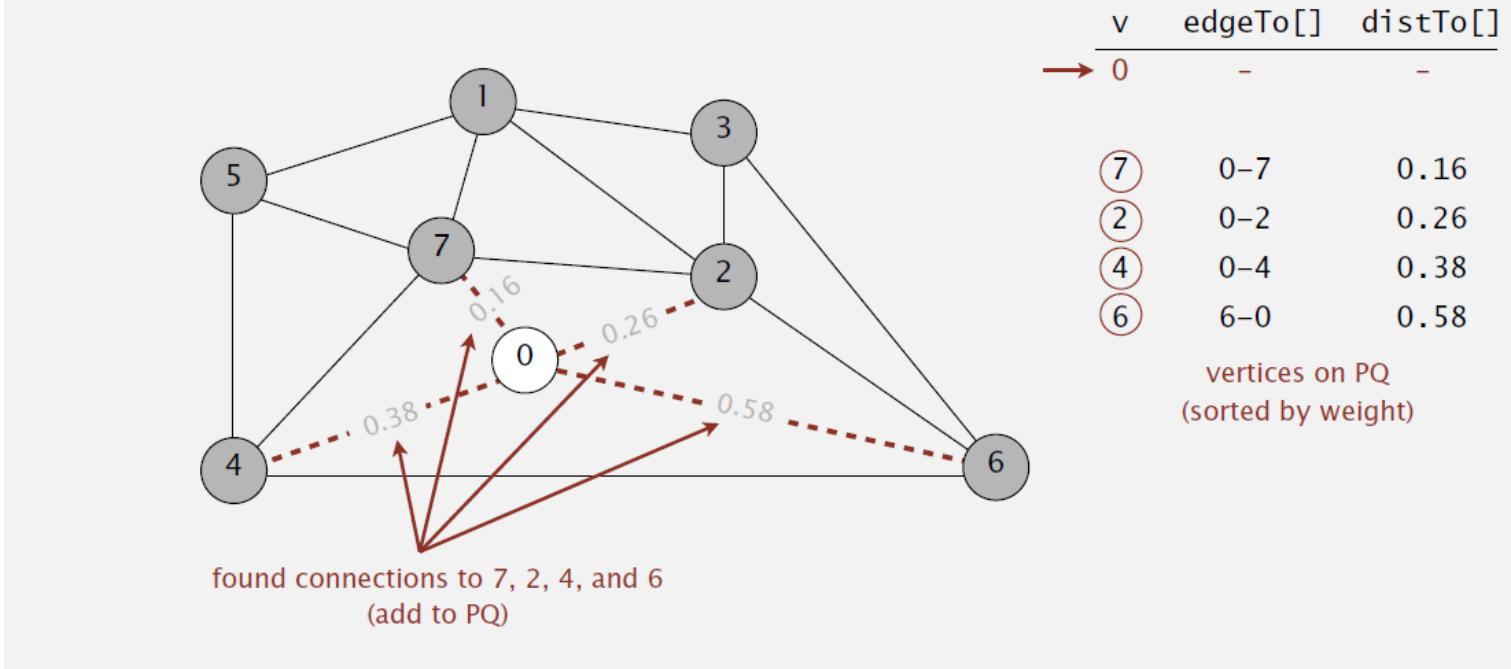
Prim's algorithm: eager implementation demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



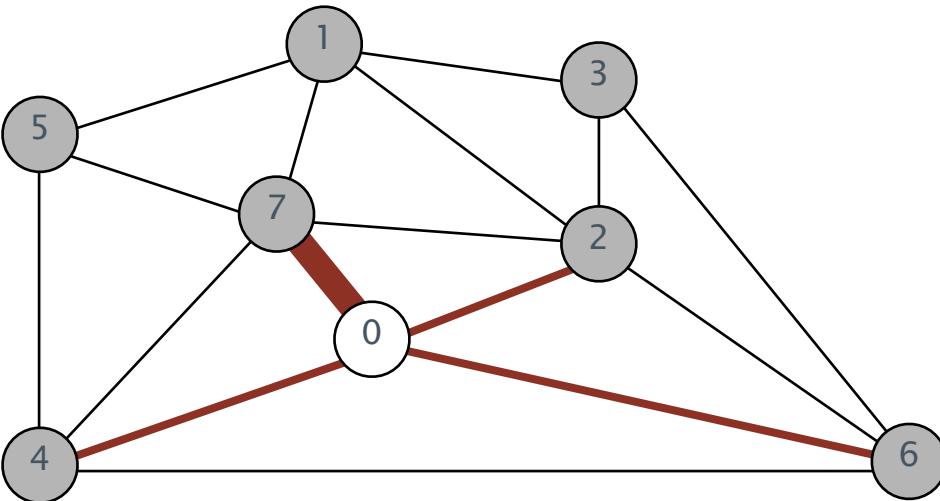
Prim's algorithm: eager implementation demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



Prim's algorithm: eager implementation demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.

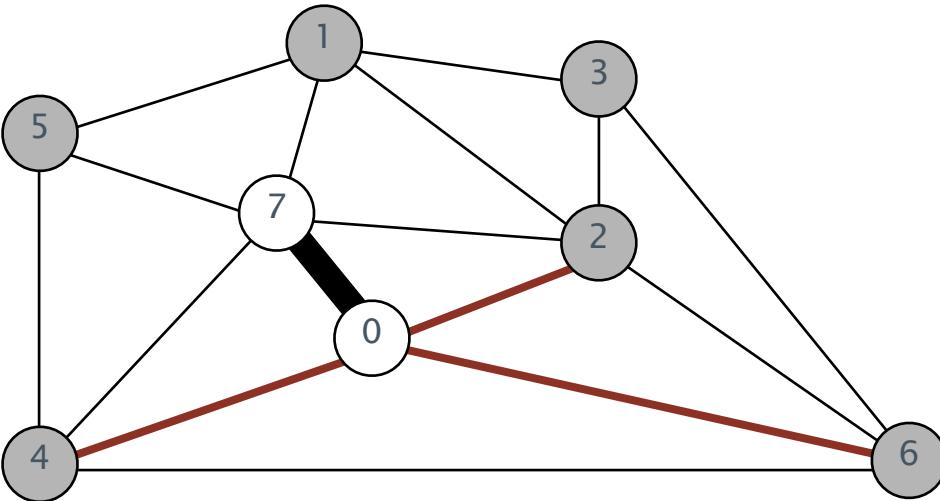


v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
2	0-2	0.26
4	0-4	0.38
6	6-0	0.58

vertices on PQ
(sorted by weight)

Prim's algorithm: eager implementation demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



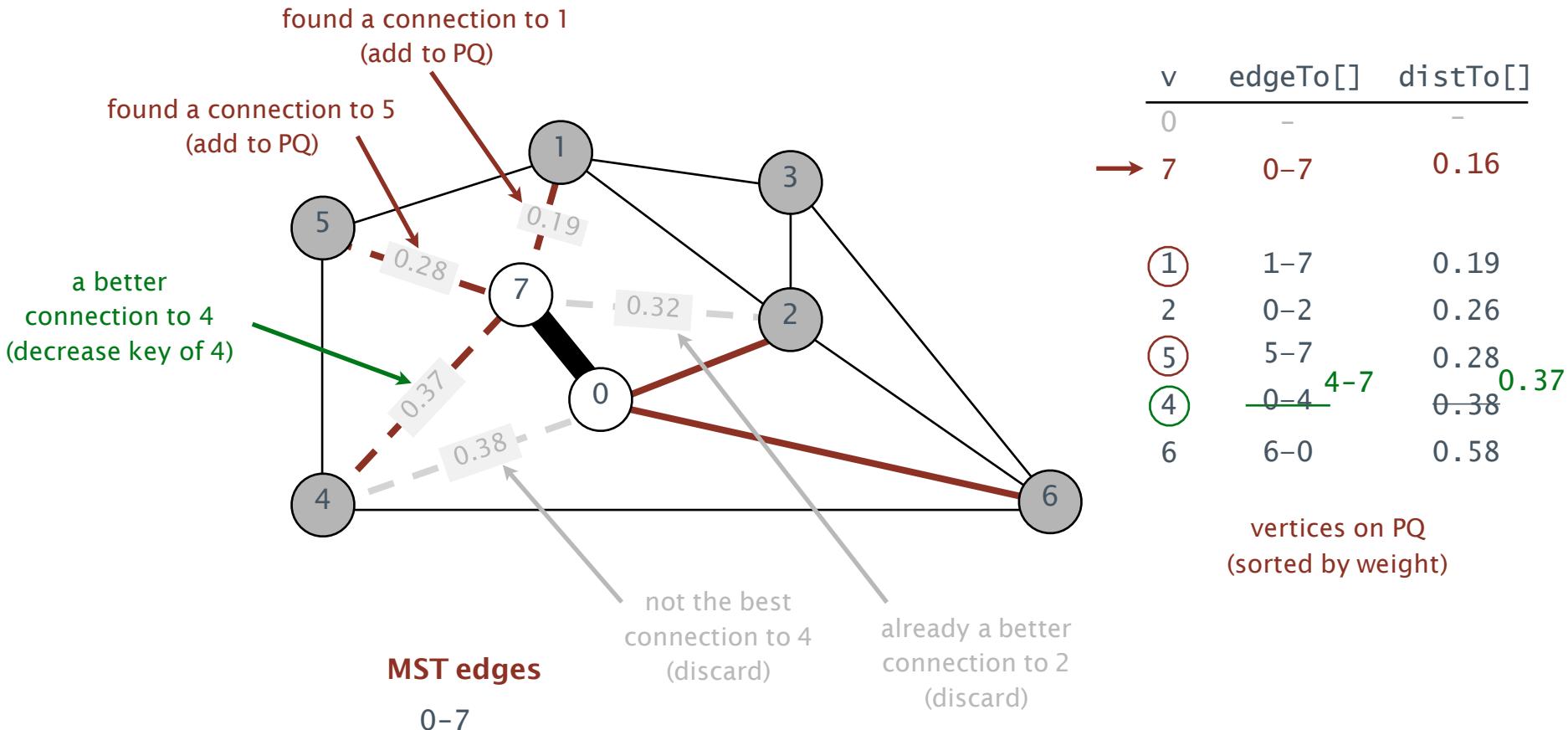
v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
2	0-2	0.26
4	0-4	0.38
6	6-0	0.58

MST edges

0-7

Prim's algorithm: eager implementation demo

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V - 1$ edges.



Indexed priority queue

Associate an index between 0 and $N - 1$ with each key in a priority queue.

- Supports **insert** and **delete-the-minimum**.
- Supports **decrease-key** given the index of the key.

```
public class IndexMinPQ<Key extends Comparable<Key>>
```

IndexMinPQ(int N)	<i>create indexed priority queue with indices 0, 1, ..., N – 1</i>
void insert(int i, Key key)	<i>associate key with index i</i>
void decreaseKey(int i, Key key)	<i>decrease the key associated with index i</i>
boolean contains(int i)	<i>is i an index on the priority queue?</i>
int delMin()	<i>remove a minimal key and return its associated index</i>
boolean isEmpty()	<i>is the priority queue empty?</i>
int size()	<i>number of keys in the priority queue</i>

Indexed priority queue implementation

Binary heap implementation. [see Section 2.4 of textbook]

- Start with same code as MinPQ.
- Maintain parallel arrays `keys[]`, `pq[]`, and `qp[]` so that:
 - `keys[i]` is the priority of i
 - `pq[i]` is the index of the key in heap position i
 - `qp[i]` is the heap position of the key with index i
- Use `swim(qp[i])` to implement `decreaseKey(i, key)`.

i	0	1	2	3	4	5	6	7	8
<code>keys[i]</code>	A	S	O	R	T	I	N	G	-
<code>pq[i]</code>	-	0	6	7	2	1	5	4	3
<code>qp[i]</code>	1	5	4	8	7	6	2	3	-

Diagram of a binary heap structure:

```
graph TD; A((A)) -- 1 --> O((O)); A -- 1 --> G((G)); O -- 4 --> R((R)); N((N)) --- O; N --- S((S)); G -- 3 --> I((I)); G -- 3 --> T((T));
```

The root node is A. Node N has priority 2 and is highlighted with a red oval. Node R has priority 8.

Prim's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap	1^\dagger	$\log V^\ddagger$	1^\dagger	$E + V \log V$

\dagger amortized

Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

MST for Digraphs?

- › Equivalent for digraphs is "minimum spanning arborescence" which will produce a tree where every vertex can be reached from a single vertex (spanning arborescence of minimum weight, optimum branching)

Greedy Algorithms

- › Applicable to optimisation problems
- › Constructs a solution through a sequence of steps, each expanding on the partially constructed solution obtained so far, until a complete solution is built
- › On each step, a choice is made which is:
 1. Feasible – has to satisfy problem constraints
 2. Locally optimal – it has to be the best local choice among all feasible choices
 3. Irrevocable – once made, it cannot be changed on subsequent steps of the algorithm
- › Greedily takes the best current option, in the hope it will add up to the overall best option
 - in some problems it does, in some it doesn't, but approximation might be good enough
- › Dijkstra's shortest path algorithm is also greedy