

Algorithms and Data Structures

Ulrike von Luxburg

Winter 2013/14

Department of Computer Science, University of Hamburg

(Version as of November 26, 2013)

Contents will be updated continuously, see the webpage for the latest version.

Table of contents

PART TWO OF THE SCRIPT

Searching

Binary Search	13
Binary search trees: basics	33
AVL Trees	59
Outlook: more search trees	86
Excursion: Inverted index for document retrieval	92

Graph algorithms

Recap: basic graph definitions	108
Data structures for representing graphs	121
Walking through graphs	128

Table of contents (2)

Depth first search	130
Application: Connected components	136
Application: Strongly connected components	138
Application: Topological sort	155
Breadth first search	169
Application: shortest paths in unweighted graphs	176
Shortest path problems	189
Single Source Shortest Paths	196
Bellman-Ford algorithm	200
Dijkstra's algorithm	212
Generic labeling method	231
All pairs shortest paths on general graphs	236
Floyd-Warshall algorithm	238
Johnson's algorithm	249
Point to Point Shortest Paths	266

Table of contents (3)

Bi-directional Dijkstra	268
Minimal spanning trees	283
Problem definition	284
Generic algorithm	289
Kruskal's algorithm, Idea	304
Union-find data structure	314
Kruskal's algorithm, revisited	332
Prim's algorithm	335

Standard Literature

- ▶ The lecture closely follows the following book:
K. Mehlhorn, P. Sanders: Algorithms and data structures: the basic toolbox. Springer, 2008.
(25 Euros via Springer mycopy)
German version of the book was supposed to be available this autumn, but got delayed until next year ☹:
K. Mehlhorn, P. Sanders, M. Dietzfelbinger: Algorithmen und Datenstrukturen. Springer, 2014. 25 Euros.
- ▶ A book I like a lot because it conveys intuitions (not available in German):
Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.
Algorithms. McGraw-Hill, 2008.
pdf preprint available online.

Standard Literature (2)

- ▶ The international standard text book on algorithms and data structures is the following:
Cormen, Leiserson, Rivest, Stein: Introduction to algorithms and data structures. 3rd edition. MIT Press, 2009.
The book also exists in German translation. Many copies available in the library, both in german and english.
- ▶ Another by-now-standard book on algorithms (english only):
Jon Kleinberg, Eva Tardos: Algorithm Design. 2005

Standard Literature (3)

For fun reading on algorithms (popular science style) I highly recommend:

- ▶ Vöcking et al., Taschenbuch der Algorithmen. Springer, 2008
(English version: Algorithms unplugged, Springer 2011)
- ▶ John MacCormick: 9 Algorithms that changed the future.
Princeton University Press, 2012

For preparing a job interview at one of the top companies:

- ▶ G. Laakmann: Cracking the coding interview. 2010.

PART TWO OF THE SCRIPT

Searching

Introduction: The searching problem

Find a particular book in a library:

[TIT] Titel (Stichwort)	Algorithms	und
[PER] Person, Autor	Cormen	und
[THM] Alle Themen		und
[PUB] Ort,Verlag (Stichwort)		
sortiert nach Erscheinungsjahr		
Erscheinungsjahr		zum Beispiel: 1948-1980 oder 1976- oder 1955
Sprache	-- Alle Sprachen --	
Land	-- Alle Länder --	
unscharfe Suche	<input type="checkbox"/>	

Introduction: The searching problem (2)

Solve a google query (find webpages that match a query):

The screenshot shows a Google search results page. The search bar contains the query "Algorithmen und Datenstrukturen" Hamburg -HAW. Below the search bar, there are tabs for Web, Images, Maps, Shopping, More, and Search tools. The 'Web' tab is selected. Below the tabs, it says "About 131,000 results (0.48 seconds)". The first result is a link to "Algorithmen und Datenstrukturen" at www.informatik.uni-hamburg.de/TGI/lehre/vl/.../AuD.... The second result is a link to "Ulrike von Luxburg - Vorlesung Algorithmen und Datenstrukturen" at www.informatik.uni-hamburg.de/ML/.../index.shtml. The third result is a link to "64-070 Vorlesung Algorithmen und Datenstrukturen - Universität ..." at <https://www.stine.uni-hamburg.de/scripts/mgrqispi.dll?...>. The fourth result is another link to "64-070 Vorlesung Algorithmen und Datenstrukturen - Universität ..." at <https://www.stine.uni-hamburg.de/scripts/mgrqispi.dll?...>.

"Algorithmen und Datenstrukturen"
www.informatik.uni-hamburg.de/TGI/lehre/vl/.../AuD... ▾ Translate this page
Arbeitsbereich THEORETISCHE GRUNDLAGEN DER INFORMATIK Algorithmen und Datenstrukturen (AD) Prof. Dr. Matthias Jantzen · PD Dr. Michael Köhler- ...

Ulrike von Luxburg - Vorlesung Algorithmen und Datenstrukturen
www.informatik.uni-hamburg.de/ML/.../index.shtml ▾ Translate this page
Algorithmen und Datenstrukturen. Fachbereich Informatik, Universität Hamburg, Wintersemester 2013/14. Organisatorisches Dozentin: Prof. Ulrike von Luxburg

64-070 Vorlesung Algorithmen und Datenstrukturen - Universität ...
<https://www.stine.uni-hamburg.de/scripts/mgrqispi.dll?...> ▾
Im Rahmen dieser Veranstaltung werden Entwurfsprinzipien & effiziente Algorithmen und Datenstrukturen vermittelt. Dabei werden eine Reihe von ...

64-070 Vorlesung Algorithmen und Datenstrukturen - Universität ...
<https://www.stine.uni-hamburg.de/scripts/mgrqispi.dll?...> ▾
Dabei werden eine Reihe von grundlegenden Algorithmen und Datenstrukturen vorgestellt, die zur Lösung häufig auftretender Teilprobleme komplexer ...

Introduction: The searching problem (3)

Goal:

- ▶ Want to construct efficient data structures to maintain a set of elements
- ▶ We want to have the following elementary operations as fast as possible:
 - ▶ Find an element with a particular key value
 - ▶ Insert a new element
 - ▶ Remove an element

Binary Search

Mehlhorn Sec. 2.5

Binary search, intuition

- ▶ Let A be an array with n elements that are sorted in increasing order.
- ▶ Given a query q , our goal is to find the (smallest) index i such that $A[i] \leq q \leq A[i + 1]$.
- ▶ Depending on the application, if the query is not in A yet, find the (smallest) index i such that if we insert the query element after i , the array remains sorted.

Binary search, intuition (2)

Intuitively, what we do is the following:

- ▶ Consider the item in the middle, element $A[n/2]$.
 - ▶ If it is smaller than q , we know that we need to continue search in the right half.
 - ▶ If it is larger than q , we continue to the left half.
- ▶ and so on ..

Binary search, intuition (3)

Slightly different scenarios:

Scenario 1:

- ▶ If query element q in the array, return **any** index with $A[i] = q$
- ▶ If query element not in the array, return **NotFound**.

Scenario 2:

- ▶ If query element q in the array, return any index with $A[i] = q$
- ▶ If *value* does not exist in the array, return the **leftmost index** i such that if we insert the value after position i to the array, the array is still sorted.

Scenario 3:

- ▶ If query element q is in the array, return the leftmost index i with $A[i] = q$.
- ▶

Binary search, pseudo-code for scenario 1

```
1 BinarySearch(A[0..N-1], value) {
2     low = 0
3     high = N - 1
4     while (low <= high) {
5         // invariants: value > A[i] for all i < low
5         // invariants: value < A[i] for all i > high
6         mid = (low + high) / 2    // round to lower integer
7         if (A[mid] > value)
8             high = mid - 1
9         else if (A[mid] < value)
10            low = mid + 1
11     else
12         return mid
13 }
14 return not_found // value would be inserted at index "low"
```

strict inequalities
↙

Binary search, pseudo-code for scenario 1 (2)

Let's run this algorithm on a couple of examples:

- ▶ Search for 5 in the following array: 1 3 5 17 18 20 26 40
- ▶ Search for 16 in the same array
- ▶ Search for 50 in the same array
- ▶ Search for 17 in the following array: 1 17 17 17 17 17 17 20

Analysis: Termination

DOES THE ALGORITHM ALWAYS TERMINATE? WHY?

Analysis: Termination (2)

Formal proof: We need to argue that the while loop ends. Assume we are in iteration i of the while loop.

- ▶ At the beginning of the while loop we have $low \leq high$ (otherwise we would not have entered the while loop).
- ▶ After line 5, we have $low \leq mid \leq high$.
- ▶ Either the loop returns with line 11. Then we are done.
- ▶ In case the loop does not return with line 11, it is in one of the two first cases of the if statement. Then, either $high$ is going to decrease (line 7) or low is going to increase (line 9) by at least 1. **Hence, in each iteration of the while loop the difference $high - low$ decreases by at least 1.**
- ▶ **So after at most n iterations of the while loop, $high - low < 0$ and the loop terminates.**



Analysis: Correctness

DOES THE ALGORITHM ALWAYS RETURN A CORRECT
(=scenario 1) RESULT?

Analysis: Correctness (2)

Yes, here comes the **Proof**:

Obviously, if the algorithm returns “mid” (line 11), then $A[mid] = value$, which is a correct result.

Need to prove: if the algorithm returns with line 13, then *value* is indeed not contained in the array.

We are now going to prove this by contraposition: **Have to show that if *value* exists in the array, then the algorithm does not return with line 13, but with line 11.**

Analysis: Correctness (3)

First step: It is straight forward to see that the following **invariances** are always maintained:

- ▶ $\text{value} > A[i]$ for all $i < \text{low}$ (strict inequality!)
- ▶ $\text{value} < A[i]$ for all $i > \text{high}$ (strict inequality!)

Analysis: Correctness (4)

Second step: Assume *value* is contained in the array.

- ▶ Know already: Invariances are true. This means that at no point of the algorithm, the element we are looking for could be left of *low* or right of *high*.
- ▶ **The only way we could potentially “miss” the array element is if either line 7 or line 9 lead to the situation $low > high$ before we have found the desired entry (because then we leave the while loop).**
 - ▶ By construction we always have $low \leq mid \leq high$ after line 7 has been executed.
 - ▶ As long as $high \geq left + 2$, we always have $low < mid < high$. In this situation, $mid - 1 \geq low$ and $mid + 1 \leq high$, so we still have $low \leq high$ after lines 7 or 9 have been executed, so we are going to enter the while loop again.

Analysis: Correctness (5)

Two critical cases where we could leave the while loop:

- ▶ $low = high$. Then also $mid = low$. By assumption, $value$ is in the array, so it has to be $A[low]$. And this is also what is going to be returned by line 11.
- ▶ $high = low + 1$. Then $mid = low$ (because we take the floor when computing mid).

Now either $A[low] = A[mid] = value$, in which case we return the correct result in line 11.

Or $A[low] = A[mid] < value$, in which case we increase low by 1, then ending up in case $low = high$ in the next iteration of the loop. This one is going to give the correct answer, as we have already seen.

- ▶ **This argument shows that if $value$ is contained in the array, then the algorithm always ends with line 11.**



Analysis: Running time

WHAT IS YOUR GUESS?

Analysis: Running time (2)

Proposition 1

Binary search has running time $O(\log n)$.

Proof:

- ▶ 2 Operations before the while loop
- ▶ Number of iterations of the while loop:
 - ▶ If n is odd, then the subarrays $A[low, \dots, mid]$ and $A[mid, \dots, high]$ contain $(n - 1)/2 \leq n/2$ elements each.
 - ▶ If n is even, then the subarrays contain $n/2 - 1$ and $n/2$ elements each.
 - ▶ The search ends at latest if the array only contains one element. This is the case after $k = \log n$ iterations.
- ▶ Within each while loop, we need 4 basic operations.
So all in all we end up with $4 \log n + 2 = O(\log n)$ basic operations.

Binary search, scenario 2

If *value* does not exist in the list, we want to know the leftmost index *i* such that if we insert the value after position *i* to the array, the array is still sorted.

```
1 BinarySearch_Left(A[0..N-1], value) {
2     low = 0
3     high = N - 1
4     while (low <= high) {
5         // invariants: value > A[i] for all i < low
5         // invariants: value <= A[i] for all i > high
6         mid = (low + high) / 2
7         if (A[mid] >= value)
8             high = mid - 1
9         else
10            low = mid + 1
11     }
12 }  
13 }
```

Variant: binary search recursively

Obviously, one can also implement binary search recursively, for example:

```
// initially called with low = 0, high = N - 1
BinarySearch_Left(A[0..N-1], value, low, high) {
    // invariants: value > A[i] for all i < low
                  value <= A[i] for all i > high
    if (high < low)
        return low
    mid = (low + high) / 2
    if (A[mid] >= value)
        return BinarySearch_Left(A, value, low, mid-1)
    else
        return BinarySearch_Left(A, value, mid+1, high)
}
```

Variant: Exponential search

Assume you don't know how large the array is (but it is still sorted).
Then:

- ▶ Compare the input query against the values with index $1 = 2^0, 2 = 2^1, 2^2, 2^3, 2^4, \dots$ until you hit an index i with $query < A[2^i]$ for the first time.
- ▶ Then you start a binary search with $start = 2^{i-1}$ and 2^i .

Easy to prove:

- ▶ Algorithm always terminates and gives the correct result.
- ▶ If $A[m] \leq query \leq A[m + 1]$, then the running time of this algorithm is still $O(\log m)$.

Remarks

- ▶ Conceptually, binary search is very easy.
- ▶ But it is very easy to make mistakes in the pseudo-code ($<$ instead of \leq or stuff like this).
- ▶ The only way to find out whether your code is correct is to formally argue using invariances.

Remarks (2)

Assumptions:

- ▶ Binary search only works if the input array is indeed sorted.
- ▶ If this condition is violated, no guarantees whatsoever can be given.
- ▶ Obviously, it is impossible to check the condition that the array is sorted in logarithmic time.

Binary search trees: basics

Literature: Cormen 12, (Mehlhorn 7)

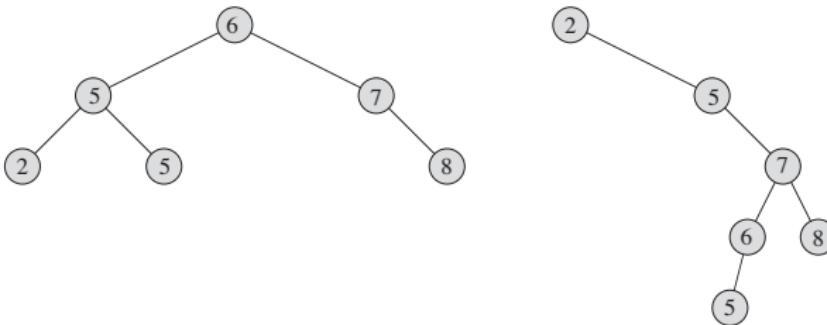
Search tree: definition

- ▶ Each element consists of four things: key value, left pointer, right pointer, parent pointer
- ▶ Vertices are arranged as a binary tree.
- ▶ The following property is satisfied for all vertices:
If y is a vertex in the tree, then
 - ▶ all vertices z in the left subtree satisfy $\text{key}(z) \leq \text{key}(y)$
 - ▶ all vertices z in the right subtree satisfy $\text{key}(z) \geq \text{key}(y)$

Note that we allow for equality both in the left and right subtree,
this is not a typo!

Search tree: definition (2)

Example: two different search trees, both contain the same numbers.



Searching an element

Want to search whether a particular query q is somewhere in the tree.

ANY IDEAS???

Searching an element (2)

Informal solution:

- ▶ Start at the root vertex.
- ▶ At every vertex y :
 - ▶ if $q = y$, return y
 - ▶ if $q > y$, descend to the right subtree
 - ▶ if $q < y$, descend to the left subtree
- ▶ If we reach the bottom of the tree and still have not found q , return an error message.

Make sure you understand why it does not lead to problems even though we have \leq and \geq in the left and right tree.

Searching an element (3)

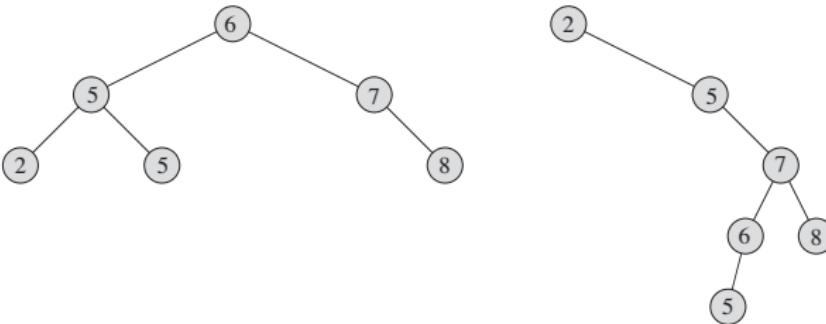
Pseudocode: x root of the tree, we search for k

TREE-SEARCH(x, k)

- 1 **if** $x == \text{NIL}$ or $k == x.key$
- 2 **return** x
- 3 **if** $k < x.key$
- 4 **return** TREE-SEARCH($x.left, k$)
- 5 **else return** TREE-SEARCH($x.right, k$)

Searching an element (4)

EXAMPLES: SEARCH FOR 5 AND FOR 8 AND FOR 3 IN THE FOLLOWING TREES:



Searching an element (5)

Observe that it does not hurt that vertices with equality can be either in the left or the right subtree (WHY)?

Searching an element (6)

RUNNING TIME?

Searching an element (7)

This operation takes $O(h)$ steps where h is the height of the tree.

Extracting minimum / maximum element from the tree

WHAT DO WE NEED TO DO TO FIND THE MINIMUM (MAXIMUM) ELEMENT IN THE TREE?

RUNNING TIME?

Extracting minimum / maximum element from the tree (2)

Minimum:

- ▶ Start in the root, always walk to the left child, until you reach a leaf.
- ▶ This leaf is the minimal element in the search tree.
- ▶ Running time is $O(h)$ where h is the height of the tree

Output all vertices as ordered list

Given a search tree, want to write all its elements into an array such that the values are ordered.

ANY IDEAS?

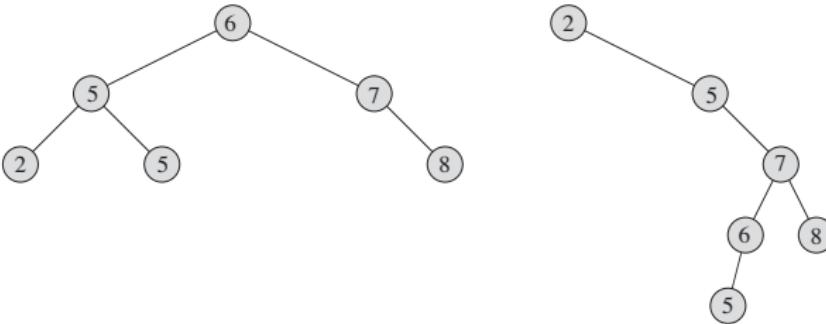
Output all vertices as ordered list (2)

INORDER-TREE-WALK(x)

- 1 **if** $x \neq \text{NIL}$
- 2 INORDER-TREE-WALK($x.\text{left}$)
- 3 print $x.\text{key}$
- 4 INORDER-TREE-WALK($x.\text{right}$)

Output all vertices as ordered list (3)

EXAMPLE: TRY IT ON THE TWO TREES FROM THE BEGINNING:



Output all vertices as ordered list (4)

Correctness: by induction over the number n of vertices in the tree

- ▶ Base case: $n = 1$ (just the root): clear
- ▶ Induction assumption: Statement is satisfied for all $m \leq m$.
- ▶ Induction step from n to $n + 1$:
 - ▶ Call the procedure at the root
 - ▶ By induction hypothesis, the left and right subtree will be printed in the correct order
 - ▶ By definition of the search tree, if we first print the left subtree, then the root, and then the right subtree, then this is going to be ordered correctly.



Output all vertices as ordered list (5)

RUNNING TIME?

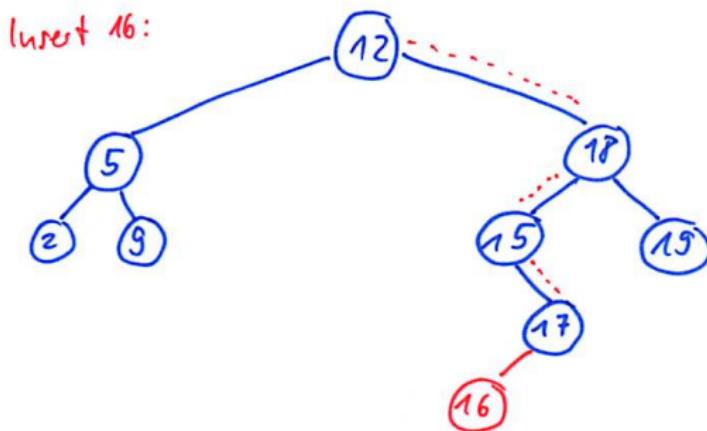
Output all vertices as ordered list (6)

Running time:

- ▶ We visit each vertex exactly once.
- ▶ In each vertex, the actual work we have to do is:
 - ▶ to print the key of this vertex
 - ▶ call two subroutines
- ▶ So the running time is $\Theta(n)$.

Inserting elements

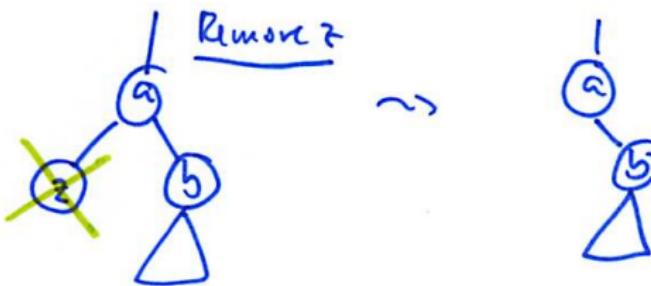
Simply walk down the tree in the obvious way and insert a new leaf:



Obviously takes $O(h)$ time as well.

Deleting an element: three cases

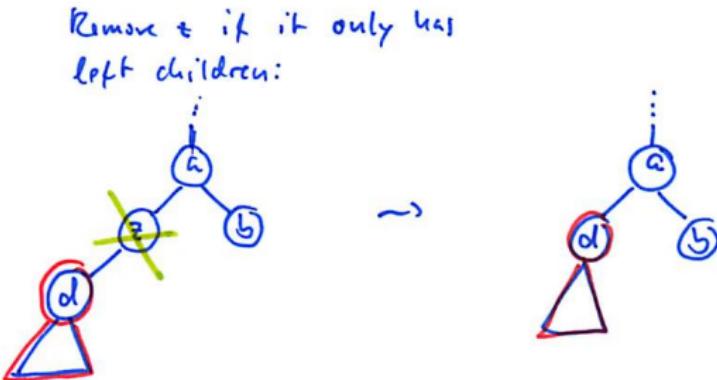
Case 1: Element has no children. Simply remove vertex.



if z has no children

Deleting an element: three cases (2)

Case 2: Element has only left (or only right) children. Simply move them one level up.

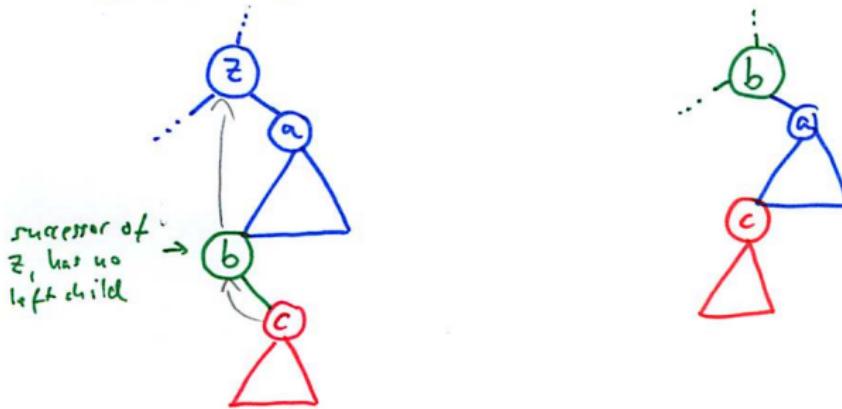


Deleting an element: three cases (3)

General case: want to remove an interior vertex z .

- ▶ Find the successor b of z (the left most vertex in the right subtree).
Note that the successor itself has no left child.
- ▶ Replace b by its right child c and z by b .

Remove z (general case):



Deleting an element: three cases (4)

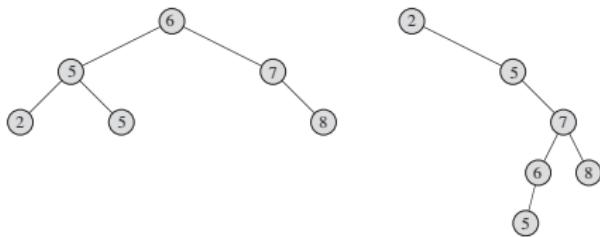
Running time:

- ▶ Finding the successor needs $O(h)$
- ▶ the rest happens in constant times (we just have to reset a couple of pointers).
- ▶ So overall: $O(h)$.

Balanced trees?

Have seen: all basic tree operations can be done in $O(h)$ time where h is the height of the tree.

Obviously it would be nice to have trees with height as small as possible $\rightsquigarrow h \approx \log n$.



Here, both trees contain the same values, but the left one has a smaller height

Balanced trees? (2)

EXERCISES:

- ▶ Describe a sequence of n input points for which the search tree has height $n - 1$.

Balanced trees? (3)

Balanced trees, two options:

- ▶ Don't do anything, but hope that the search tree won't be too imbalanced.
 - ▶ For example, one can prove that if we start with an unsorted array, then the average height of the corresponding search tree is $O(\log n)$ (where the average is over all possible permutations of the array). ☺(Proof: see Cormen Sec. 12.4)
 - ▶ However, little is known if we mix delete / insert operations, no average case analysis ... ☹
- ▶ Apply rebalancing operations to maintain a balanced tree.
 - ▶ See next section.

AVL Trees

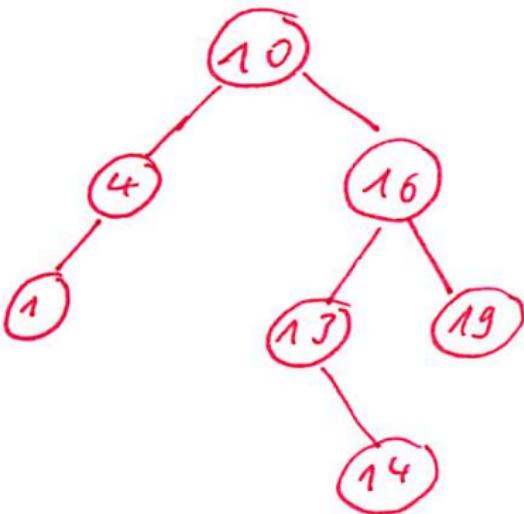
Not in any of the standard text books we use 😐

Section 5.2.1. of Ottmann / Widmeyer: Algorithmen und Datenstrukturen.

Original paper: **A**delson-**V**elskii, **L**andis: An algorithm for the organization of information. Soviet Mathematics Doklady, 1962.

Balanced search tree

We say that a **binary search tree is balanced** if at every node, the height of the left and right subtree differs by at most 1.



Height of a balanced search tree

Proposition 2

The height of a balanced search tree with n vertices is $O(\log n)$.

Proof.

- ▶ Assume we are given a balanced tree of height h .
- ▶ Denote by $N(h)$ the minimal number of vertices in a balanced binary tree of height h .
- ▶ The root has two subtrees.
 - ▶ One of the subtrees has to have height $h - 1$ (otherwise the whole tree would not have height h)
 - ▶ Then by the balancing condition, the other subtree has height at least $h - 2$.

Height of a balanced search tree (2)

- So we get

$$N(h) = N(h - 1) + N(h - 2) + 1$$

DOES IT RING A BELL?

Height of a balanced search tree (3)

Looks pretty much like the Fibonacci sequence ...

We immediately see:

$$N(h) > F_h$$

where F_h is the h -th Fibonacci number. Thus by our knowledge on F_h :

$$N(h) > F_h \geq 2^{h/2} \implies h \leq 2 \log(N)$$

Height of a balanced search tree (4)

Alternative analysis, by foot:

$$\begin{aligned}N(h) &= \underbrace{N(h-1)}_{=N(h-2)+N(h-3)+1} + N(h-2) + 1 \\&> 2N(h-2)\end{aligned}$$

Applying the same trick recursively leads to

$$N(h) > 2^2 N(h-4) > 2^3 N(h-6) \dots > 2^{h/2}$$

which implies

$$h < 2 \log N(h)$$



AVL tree Intuition

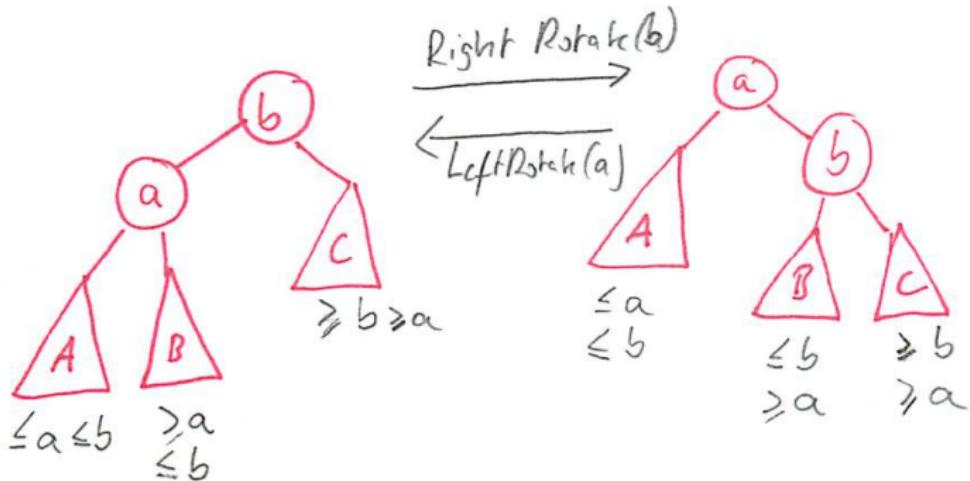
For searching, a balanced tree is pretty much optimal:

- ▶ Minimal height of a binary search tree is achieved for a full tree, $h = \log n$.
- ▶ A balanced search tree satisfies $h \leq 2 \log n$, which is nearly optimal (up to the factor of 2).
- ▶ We now want to maintain a balanced search tree.

WHICH ARE THE DIFFICULT OPERATIONS?

Basic operation: rotation

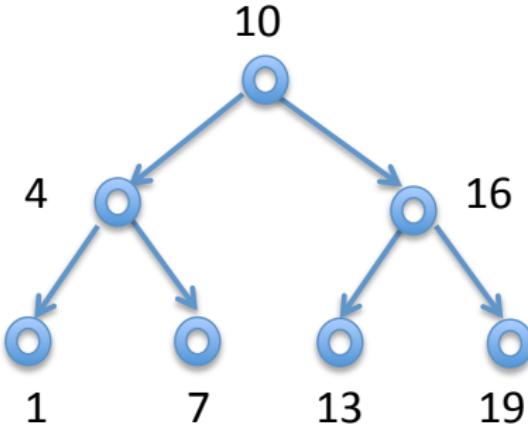
Basic operation, needed to restore the balance at a particular vertex (see later):



Note that one rotation operation just needs $O(1)$ time.

Inserting an element

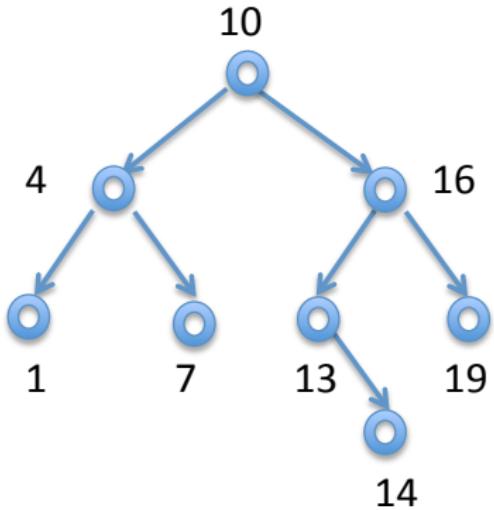
Start with a properly balanced tree.



- ▶ First insert an element “as usual”
- ▶ Now walk up towards the root and check whether the balancing conditions are still satisfied.

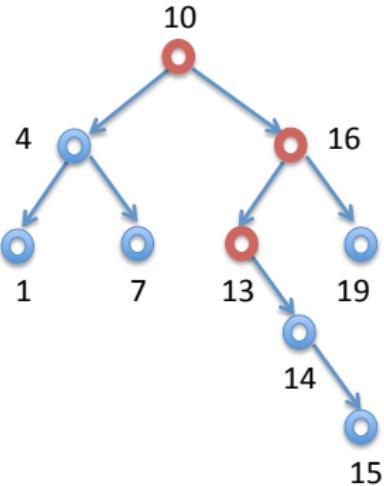
Inserting an element (2)

Good case: all vertices up to the top are balanced. Then we are happy.



Inserting an element (3)

Bad case: When we walk up towards the root, we encounter a vertex which is not balanced, that is we have height difference 2 between left and right subtree.



We want to fix this using rotations. Will see: two balancing operations are always enough.

Fixing imbalance at one vertex

- ▶ Walk up the path from the inserted vertex to the root.
- ▶ Stop at the first imbalanced vertex.
- ▶ Now we can be in two different scenarios.

Fixing imbalance at one vertex (2)

Situation 1:

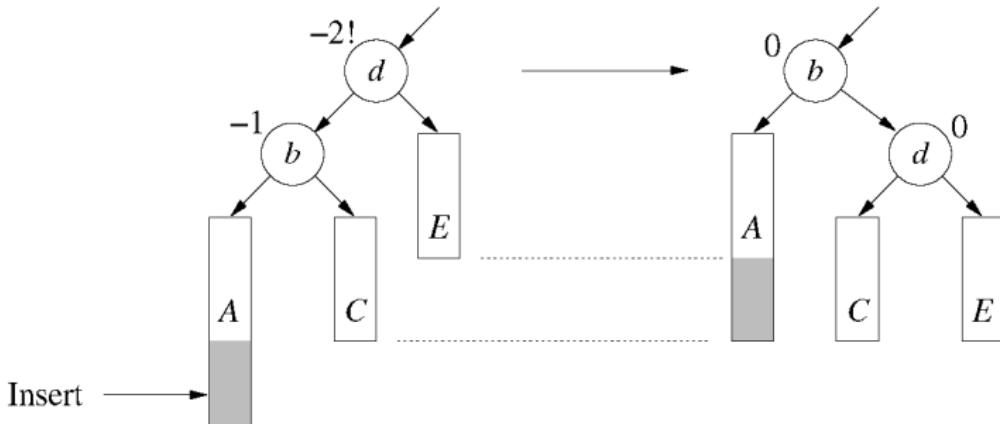


Figure 22: Single rotation.

Fixing imbalance at one vertex (3)

Situation 2:

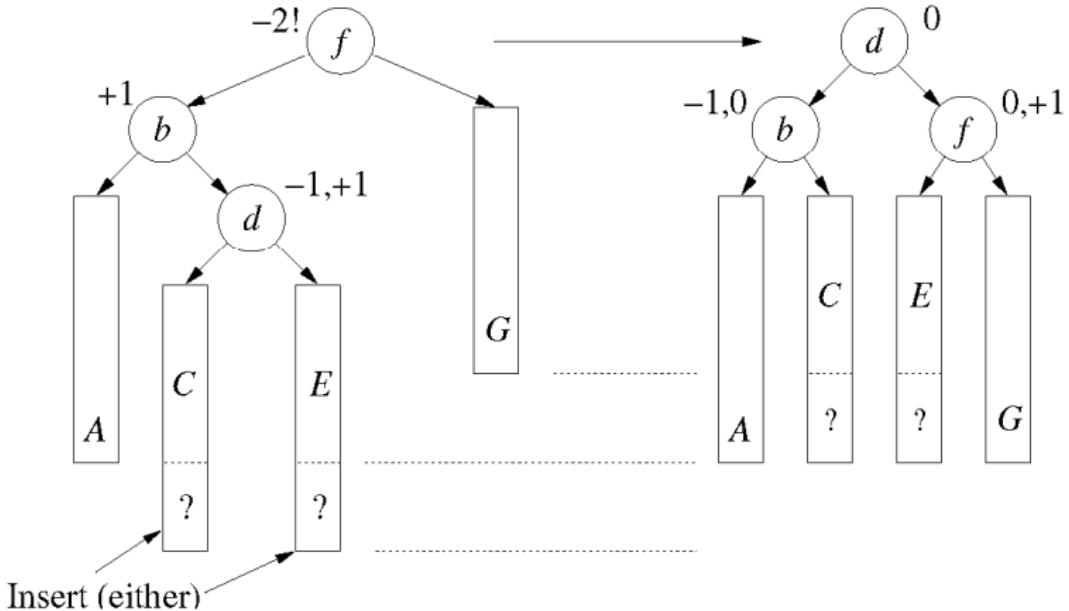


Figure 23: Left-right double rotation.

Fixing the other imbalances

By the rotation operations just discussed before, we can fix the imbalance at the first imbalanced vertex we encounter.

Somewhat surprisingly, this also fixes any other imbalance that could have been on the path towards the root:

- ▶ Let h be the height of the subtrees below b before insertion. Then after insertion and rotation, the subtrees have again height b . So for any vertex that has b as child, the height differences did not change.
- ▶ Similarly, the height of the tree below d is the same before insertion and after insertion and rotation.
- ▶ So we do not need to walk up the tree and check for the balancedness of the other vertices on the path to the root, for them everything looks as it was before.

Fixing the other imbalances (2)

height before insertion = height after insertion + rotation

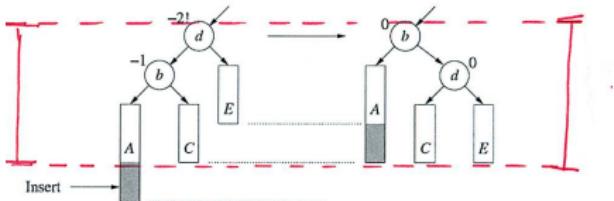


Figure 22: Single rotation.

height before insertion = height after insertion + rotation

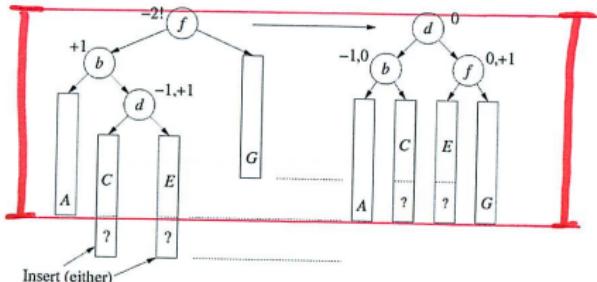
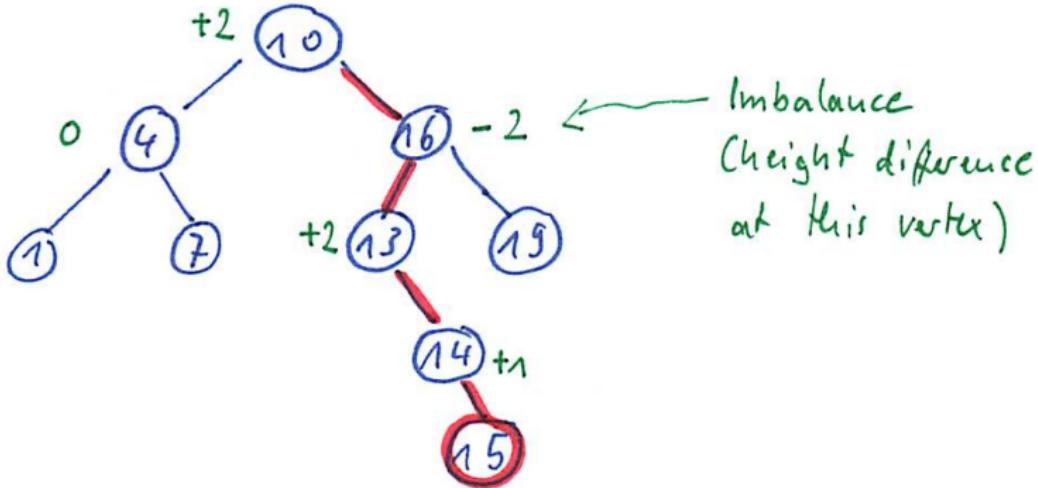


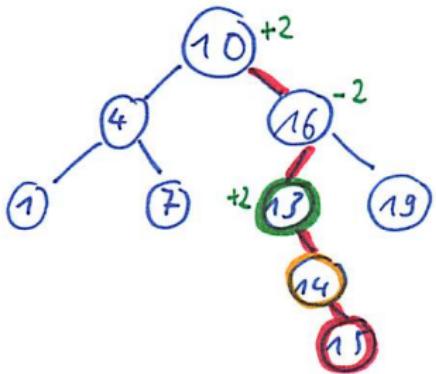
Figure 23: Left-right double rotation.

Fixing the other imbalances (3)

Example:



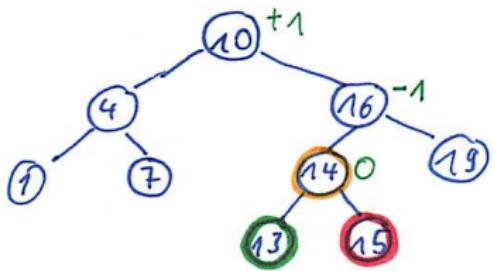
Fixing the other imbalances (4)



Walking up from 15 towards the root, 13 is the first imbalanced vertex we find.

Note: at vertex 13 we are in situation 1 (with empty A, C, E).

Fixing the other imbalances (5)



After performing the rotation,
all imbalances are fixed
(even at vertices above).

Fixing the other imbalances (6)

Formal reason:

- ▶ After rebalancing, the corresponding subtree has the same height as it had been before.
- ▶ So the imbalance of any vertex above the current one is as it had been before insertion, hence it is correct.

Running time of insertion

- ▶ Standard insertion to a search tree: $O(\log n)$
- ▶ Walk up the tree to the check balancedness: $O(\log n)$.
- ▶ For the first imbalance you find, perform at most two rotations, each $O(1)$.
- ▶ Done.

So to insert an element to an AVL and maintain the balancing property takes $O(\log n)$.

Deleting an element

Situation 1:

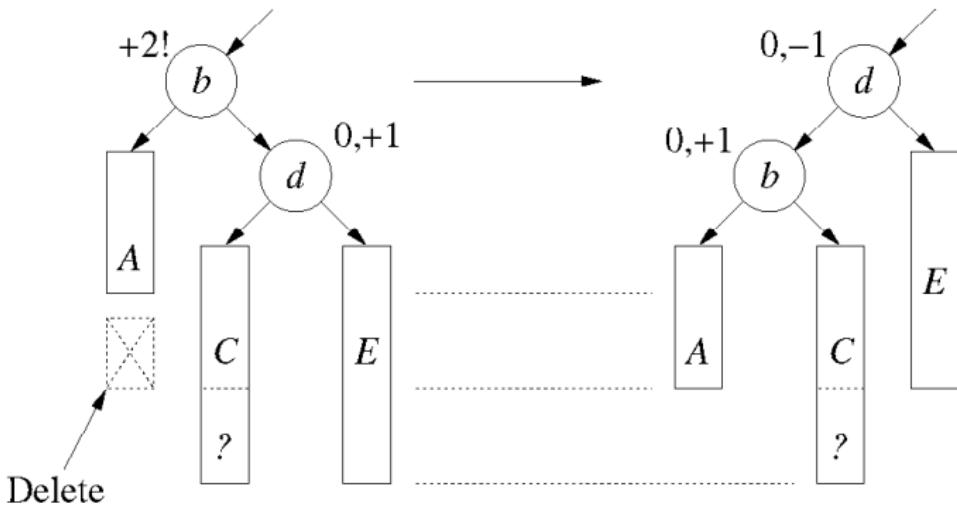


Figure 24: Single rotation for deletion.

Deleting an element (2)

Situation 2:

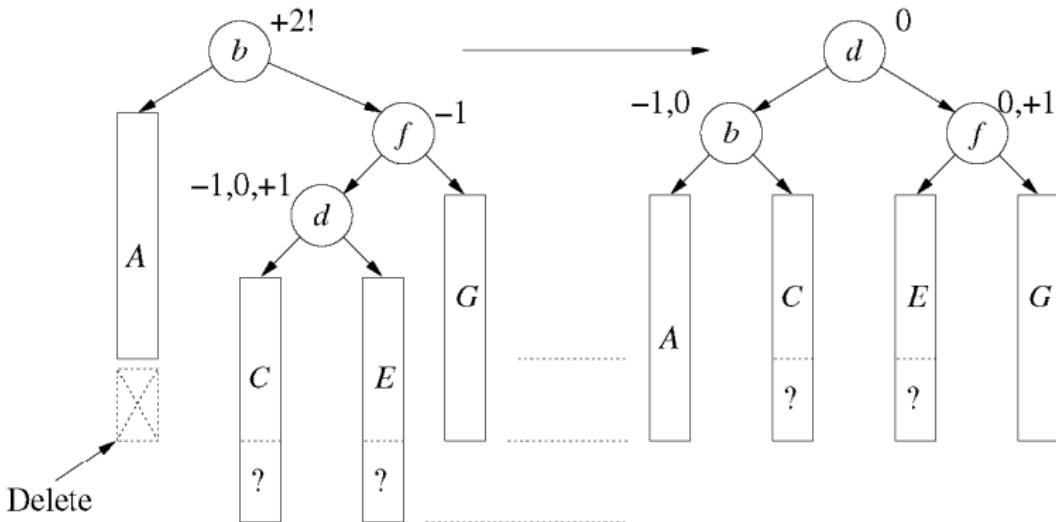


Figure 25: Double rotation for deletion.

Deleting an element (3)

A whole deletion process can require rotations for all vertices along the insertion path. Example:

Deleting an element (4)

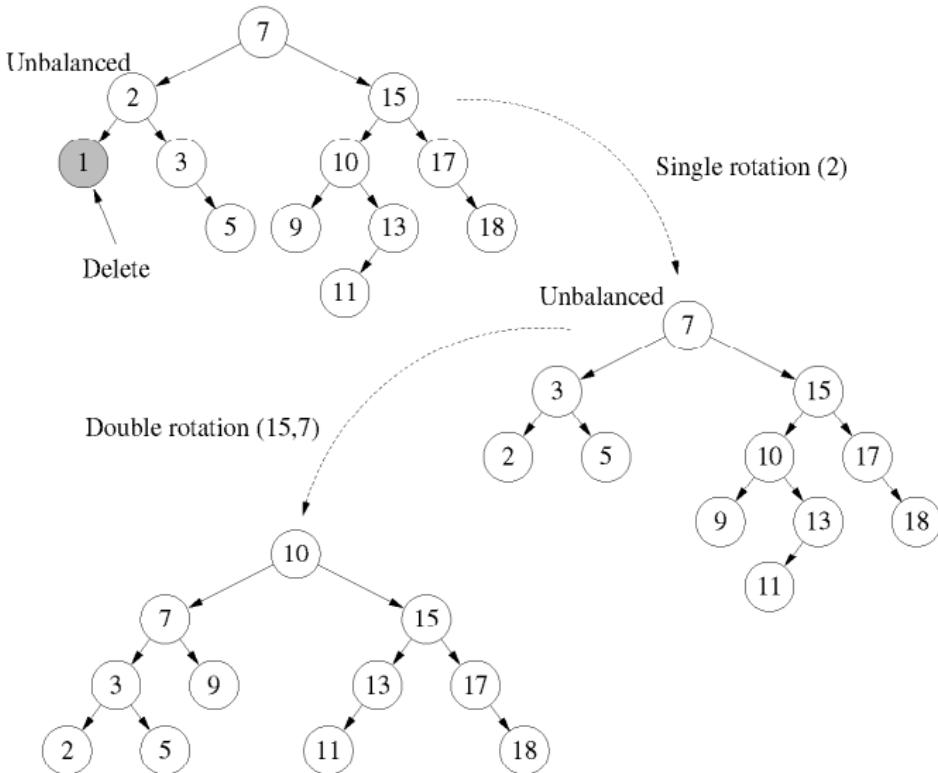


Figure 26: AVL Deletion example.

Final comments

- ▶ We omit the formal proofs, they are not difficult but length (many different cases to distinguish).
- ▶ In order to implement AVL trees, each node stores as an additional attribute its height (or just the difference of the height of its subtrees, as an integer in $\{-1, 0, 1\}$). Need to maintain it throughout all the rotations.

History: AVL tree

The first tree data structure for searching was the “AVL” tree:

- ▶ **Adelson-Velskii, Landis:** An algorithm for the organization of information. Soviet Mathematics Doklady, 1962.

Outlook: more search trees

More search trees

- ▶ There exists a large variety of search trees, all with slightly different properties ... see Section 7.7. for some references.
- ▶ General trade-off:
Harder balancing constraints \implies fast retrieval but slower insertion/removal
- ▶ To save costs, one might tolerate a certain unbalancedness, and just rebalance the tree if the unbalancedness becomes too large.

More search trees (2)

Additionally to the basic operations, we might want to support some of the following operations:

- ▶ Find min / max
- ▶ Concatenate / merge two search trees
- ▶ Split two search trees
- ▶ ...

Red black tree

- ▶ Is another balanced binary search tree architecture.
- ▶ Vertices are either “red” or “black”.
- ▶ A red vertex has only black children.
- ▶ At any vertex x : all paths from x to leaves contain the same number of black vertices.

One can prove:

- ▶ The height of such a tree is $O(\log n)$. Hence it is approximately balanced.
- ▶ Inserting and deleting vertices can be done in $O(\log n)$ (but it is reasonably complicated because we have to make sure that the red-black-properties are fixed after changing the tree).

Red black tree (2)

Text book references: Cormen

Original references:

- ▶ First invented in:
Rudolf Bayer. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica* 1 (4): 290–306, 1972.
- ▶ Naming convention “red” and “black”:
J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees”. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*. 1978.

Splay tree

Invented by D. Sleator and R. Tarjan, 1985.

General idea:

- ▶ Whenever we access an element x in the tree, we perform a “splay operation”. This operation moves x to the root of the tree.
- ▶ In this way, elements that are often accessed sit close to the root (fast access when we need it the next time).
- ▶ We also call splay operations after inserting an element (to move it to the root) and after deleting an element (then we apply the splay operation to the parent of the deleted vertex).

If element e is accessed with probability p , then over time the tree will be reshaped to allow for access to e in time $O(\log(1/p))$.

Excursion: Inverted index for document retrieval

Literature:

- ▶ Büttcher, Clarke, Cormack: Information Retrieval, 2010.
Chapter 4 (see paper repository)
- ▶ Zobel, Moffat: Inverted files for text search engines. ACM Computing Surveys, 2006. (see paper repository)

The retrieval problem

Given a collection of documents (say, all html pages in the web) and a query for a search term (say, “Hamburg”), find all pages that contain the query term.

- ▶ Obviously, if the system is supposed to scale we cannot start searching through all documents at query time.
- ▶ We need an efficient data structure that did the “scanning” already.
- ▶ Building the data structure is allowed to be somewhat expensive, but once we have it, queries should be answered very fast.
- ▶ Ideally, we would like to update the data structure online as the contents of the document collection changes.

Inverted index, idea

Like the index in a book:

- ▶ Maintain a dictionary of all words that occur in the whole collection of documents
- ▶ For each word, store a list of document IDs of all documents that contain the term.

Inverted index, idea (2)

Example:

Document collection:

<u>Id</u>	<u>Document</u>
1	It is very cold today.
2	I came by bike.
3	There is a cold wind.

Dictionary

<u>Terms</u>	<u>Occurrences</u>
It	1
is	1, 3
very	1
cold	1, 3
today	1
I	2
came	2
by	2
bike	2
there	3
:	

Data structure for storing the index

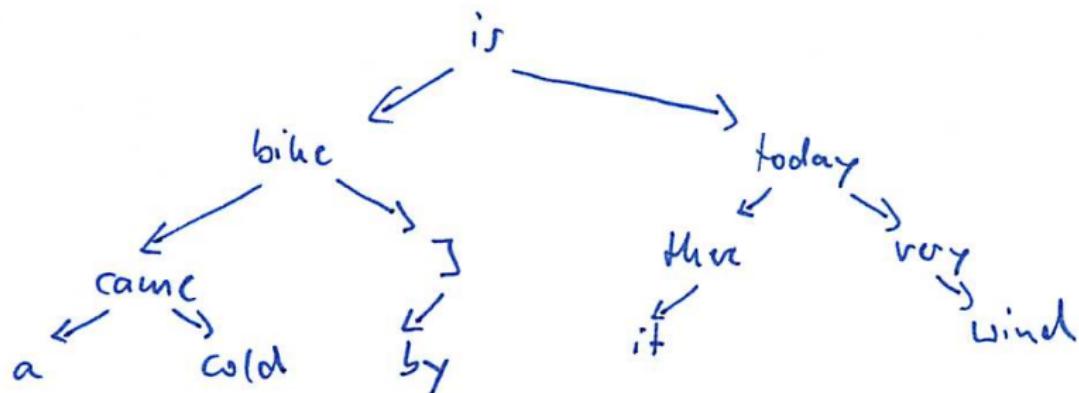
What would be a good data structure to store an inverted index?

ANY IDEAS?

Data structure for storing the index (2)

Sort-based dictionary:

- ▶ The terms are stored in a balanced search tree.
- ▶ At query time, we search the tree for the occurrence of the search term.
- ▶ Advantage: also supports prefix queries of the type “Ham*”.



Data structure for storing the index (3)

Hash-based dictionary:

- ▶ We organize the search terms in a hash table, where conflicts are resolved by chaining.
- ▶ At query time, we first evaluate which is the correct hash bin of the query, and then we look through all corresponding entries to find (or not find) the occurrences of the query term in the documents.
- ▶ As a rule of thumb, the hash table should grow linearly with the number of terms in the dictionary, to avoid too costly collisions.

Data structure for storing the index (4)



Data stored for each term occurrence

The actual information stored for each of the items is at least: the term, the IDs of the documents.

Often it is useful to even store more things such as

- ▶ term frequency: how often does the term occur in the document?
- ▶ where exactly does the term occur in the document (positions of the terms in the document)

Answering queries

Single term query:

Easy: just look in your data structure to see in which of the documents the term occurs.

Multiple term query: Term1 AND Term2:

- ▶ Look up the IDs for Term 1, and the ones for Term 2, as usual.
- ▶ Build the intersection of the two lists of IDs
Remark: this has to be done efficiently ...

Phrases like “Universität Hamburg”:

- ▶ First proceed as in the multiple term query.
- ▶ Then also evaluate the positions at which the two query terms occur.
Remark: if this has to happen efficiently, it is not straight forward.

Answering queries (2)

Ranking the search results:

- ▶ It is crucial to present the search results to the user such that the “most important” hits appear at the top of the list of results.
- ▶ The problem of rearranging the list of matches according to how important they are is called “ranking”.
- ▶ Google has been founded on the idea of a new ranking algorithm: pagerank.
- ▶ We won’t have time to discuss the details...

Optimizing the index

To build an efficient system, it is crucial to **adapt the data structure to the type of data you are dealing with.**

- ▶ In natural language, the vast majority of terms comes from a list of, say, 10.000 words.
- ▶ So we need to make sure that access to these 10.000 words is very fast.
- ▶ If this is the case, the average performance of the system is going to be fast as well.

Optimizing the index (2)

Adapting hash tables:

- ▶ Want to make sure that frequent query terms are at the beginning of the list that contains the collisions.
- ▶ **Insert-at-back:** Whenever you insert an element, insert it to the end of the list. Rare words usually do not occur early in the index construction, so the rarer the word is, the more at the back of the list it tends to be.
- ▶ **Move-to-front:** Whenever a query finds a term, move the corresponding term to the beginning of the collision list. In this case, frequently asked queries remain at the beginning of the list, rare queries at the end of the list.

Optimizing the index (3)

Adapting search trees:

- ▶ Make sure that frequent search terms are at the top of the tree.
- ▶ Whenever a query term has been accessed, perform a sequence of operations that move it closer to the top of the tree.
- ▶ Not so obvious how to do this, look up “Splay trees” for an example.

Challenge: scaling!

- ▶ Obviously, it is a huge challenge to build such an inverted index for huge systems such as the internet.
- ▶ All the components have to be saved and maintained in a highly distributed fashion.
- ▶ Details are out of the scope of this lecture. If you are interested, you could start to read the references I mentioned at the beginning of this section.

Graph algorithms

Recap: basic graph definitions

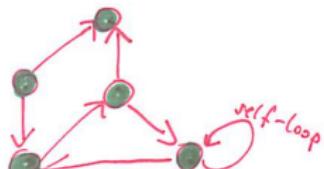
Recap: graphs

A graph $G = (V, E)$ consists of vertices $v \in V$ and edges $e \in E$.

- edges can be **directed** or **undirected**
- edges can be **weighted**, that is each edge $e = (u, v)$ has a weight $w(u, v)$.
- an edge $e = (u, v)$ is called a self-loop if $u = v$



undirected



directed

Recap: graphs (2)

Notation:

- ▶ Often we treat unweighted graphs as a special case of weighted graphs where all edge weights $w(u, v)$ are either 0 (no edge between u and v) or 1 (undirected edge exists between u and v).
- ▶ We say that u is **adjacent** to v if there exists an edge between u and v .
- ▶ In an undirected graph, we write $u \sim v$ if u is adjacent to v .
- ▶ In a directed graph we write $u \rightarrow v$ if there is a directed edge from u to v .

Recap: graphs (3)

In an undirected graph, the **degree of a vertex** v is defined as

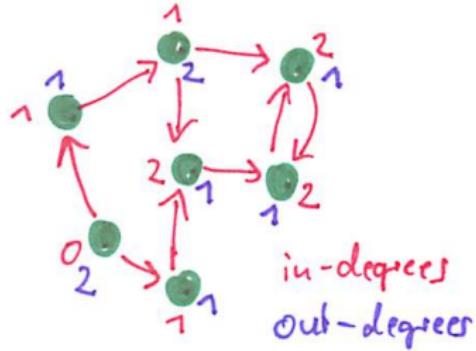
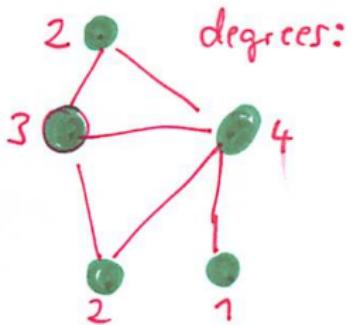
$$d_v := d(v) := \sum_{v \sim u} w_{vu}$$

In a directed graph, we define the **in-degree** and the **out-degree**:

$$d_{in}(v) = \sum_{\{u : u \rightarrow v\}} w(u, v)$$

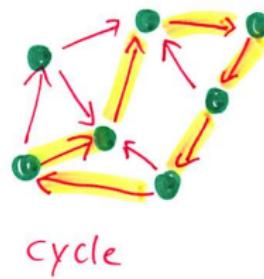
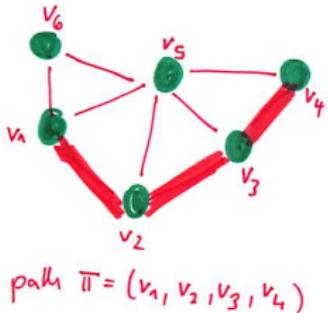
$$d_{out}(v) = \sum_{\{u : v \rightarrow u\}} w(v, u)$$

Recap: graphs (4)



Recap: graphs (5)

- ▶ A (directed) **path** in a graph is a sequence of vertices v_1, \dots, v_k such that there is a (directed) edge between v_i and v_{i+1} for all $i = 1, \dots, k - 1$.
- ▶ A path is called a **cycle** if $v_1 = v_k$, that is it ends in the vertex where it started from.

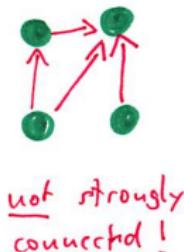
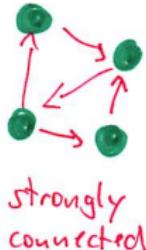
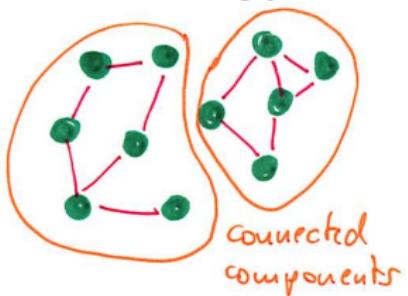


Note: in an undirected graph, walking an edge back and forth is not considered a cycle (in undirected graphs, a cycle has to contain at least 3 edges).

- ▶ A path is called **simple** if each vertex occurs at most once.

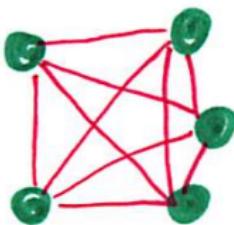
Recap: graphs (6)

- ▶ An undirected graph is called **connected** if for all $u, v \in V$, $u \neq v$ there exists a path from u to v .
- ▶ A directed graph is called **strongly connected** if for all $u, v \in V$, $u \neq v$ there exists a directed path from u to v AND a directed path from v to u .
- ▶ A **connected component** of an undirected graph is a maximal, connected subset of V .
- ▶ A **strongly connected component** of a directed graph is a maximal, strongly connected subset $A \subset V$.



Particular graphs

Complete graph: There exists an edge between all pairs of vertices $u, v \in V$.



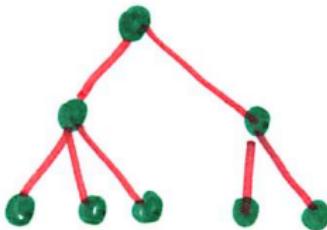
complete graph

Note: some people define the complete graph with self-loops, other people define it without self-loops.

Particular graphs (2)

Acyclic graph: A graph is called acyclic if it does not contain any cycles.

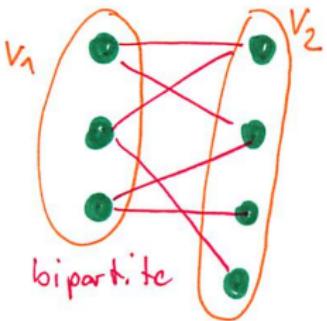
A **(undirected) tree** is an undirected, connected, acyclic graph.



Any acyclic undirected graph can be represented as a **forest**, a collection of trees.

Particular graphs (3)

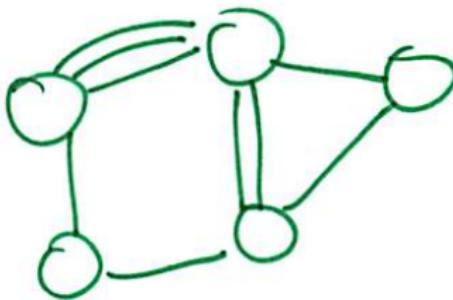
A graph is called **bipartite** if its vertex set V can be decomposed into two disjoint subsets $V = V_1 \cup V_2$ such that all edges in E are only between V_1 and V_2 , but not within V_1 or within V_2 .



Particular graphs (4)

Sometimes people consider graphs that can have several edges between vertices. Such graphs are called **multi-graphs**.

Usually, multi-graphs don't have edge weights, they can be directed or undirected.



In most cases, we can replace multi-graphs by weighted graphs.

Particular graphs (5)

Two somewhat vague notions:

- ▶ A graph is called **dense** if it has “very many edges”. The definition of “very many” depends on the current context, often it is used if the number of edges is $\Theta(n^2)$.
- ▶ A graph is called **sparse** if it has only “few edges”, for example each vertex has a very small degree compared to n . Depending on the context, a graph might be called sparse if its number of edges is in $O(n \log n)$ or even in $O(n)$.

Particular graphs (6)

Finally, general conventions:

- ▶ In many papers and books, the authors use the convention that *n* denotes the number of vertices and *m* the number of edges.
- ▶ Sometimes authors also use $|V|$ and $|E|$.

Data structures for representing graphs

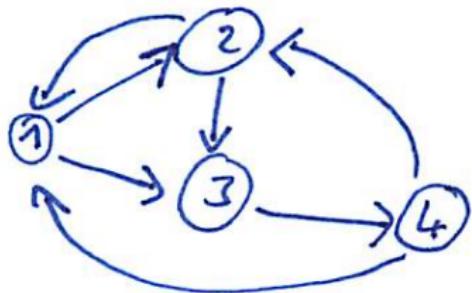
Literature: Mehlhorn Sec. 8

Representing a graph

EVERYBODY: GIVEN A GRAPH, HOW WOULD YOU STORE IT???

Adjacency matrix

- ▶ **Adjacency matrix A :** an $n \times n$ matrix (where n is the number of vertices) that contains entries $a_{ij} = 1$ if there is a directed edge from vertex i to vertex j and $a_{ij} = 0$ otherwise.
- ▶ If the graph is weighted, then the adjacency matrix contains the weights of the edges, that is $a_{ij} = w_{ij}$. By default, $w_{ij} = 0$ if there is no edge between i and j .
- ▶ To implement it, we can simply use n arrays of length n each.



Adjacency matrix:

0	1	1	0
1	0	1	0
0	0	0	1
1	1	0	0

Adjacency matrix (2)

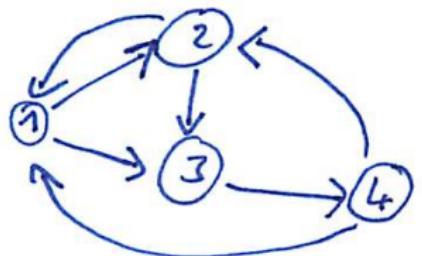
- ▶ Advantage of this representation: we can test in time $O(1)$ whether two particular vertices are adjacent.
- ▶ Disadvantage: need space n^2

When to use it:

- ▶ This representation is only useful if the graph is dense (WHY???)
- ▶ In particular circumstances, it might also be useful for sparse graphs (for example if we want to use tools from linear algebra)

Adjacency arrays

- ▶ For each vertex v , we store a little array that contains the target vertices of all edges adjacent to v
- ▶ We concatenate all these arrays to one long array of length $|E|$.
- ▶ To know which target vertices belong to which starting vertex, we additionally store the starting positions of the subarrays of all the vertices.



Adjacency array:

2 3 3 4 1 2

Edge array

1 1 3 4 5

Starting positions

Adjacency arrays (2)

- ▶ Needs $|V| + |E| + O(1)$ units of storage (space efficient!)
- ▶ Can take up to $O(n)$ time to test whether two particular vertices are adjacent (WHY?)
- ▶ Unsuitable if the graph is dynamic (WHY?)

Adjacency lists

For each vertex, we store a list of all outgoing edges (for some applications, it also makes sense to store both incoming and outgoing edges).

- ▶ Needs about m units of storage.
- ▶ Easy to add / remove edges.
- ▶ Testing whether two particular vertices are adjacent can take up to $O(n)$ time (WHY???)

Walking through graphs

The problem

We want to “walk” through a graph in a systematic way.

Example:

- ▶ We want to send a message to all sensors in a sensor network.
- ▶ We want to find out whether there exists a path from A to B in a road network.
- ▶ We want to crawl all webpages.

EVERYBODY: TAKE A COUPLE OF MINUTES. TRY TO COME UP WITH A STRATEGY TO WALK THROUGH A GRAPH SO THAT YOU VISIT EACH VERTEX (AT LEAST / AT MOST) ONCE.

Depth first search

Literature: Cormen 22.3; Mehlhorn Sec 9; (Dasgupta 3.2)

Depth first search — idea

Given any (directed or undirected) graph $G = (V, E)$.

Goal: Explore and traverse the whole graph

Algorithm: Depth first search (DFS)

General idea: Starting at one arbitrary vertex, we jump to one of its neighbors, then one of his neighbors, and so on. We take care that we never visit a vertex twice. Once the current chain ends, we backtrack and walk along another chain.

Depth first search — pseudo code

DFS(G)

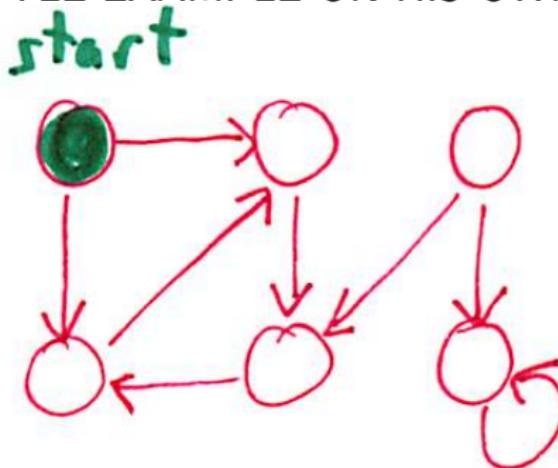
```
1 for all  $u \in V$ 
2    $u.\text{color} = \text{white}$  # not visited yet
3 for all  $u \in V$ 
4   if  $u.\text{color} == \text{white}$ 
5     DFS-Visit( $G, u$ )
```

DFS-Visit(G, u)

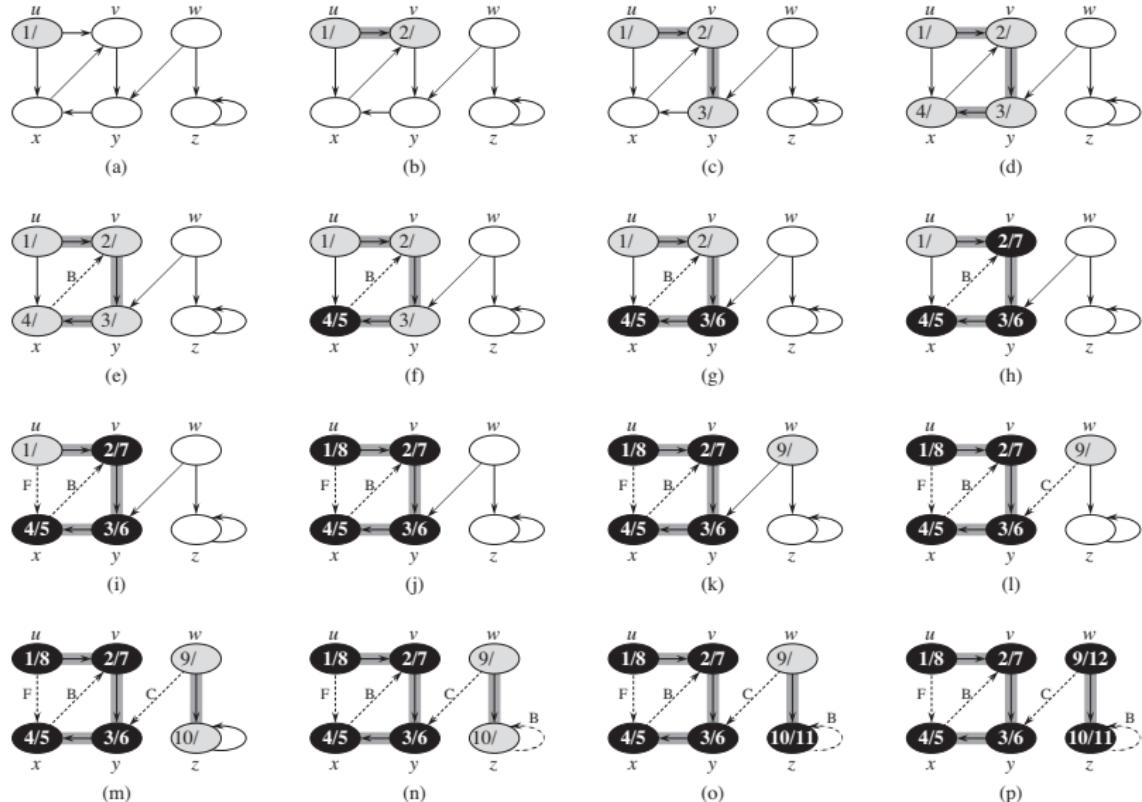
```
1  $u.\text{color} = \text{grey}$  # grey: in process
2 for all  $v \in \text{Adj}(u)$ 
3   if  $v.\text{color} == \text{white}$ 
4      $v.\text{pre} = u$  # Remember where we came from, just for analysis
5     DFS-Visit( $G, v$ )
6    $u.\text{color} = \text{black}$  # black: we are done
```

DFS — an example

EVERYBODY SHOULD TRY TO RUN DFS ON THE FOLLOWING LITTLE EXAMPLE ON HIS OWN:



DFS — an example (2)



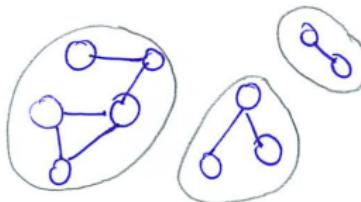
DFS — properties

- ▶ The algorithm visits each vertex of the graph exactly once (why?)
- ▶ The algorithm travels along each edge of the graph at least once (why?).
- ▶ The running time of the algorithm is $O(|V| + |E|)$ (why?)

Application: Connected components

Connected components

Recall: In an undirected graph, a **connected component** is a maximal subset A of vertices such that there exists a path between each two vertices $u, v \in A$.



Observe: if we start a DFS in a vertex u , then the tree discovered by $\text{DFS}(G, u)$ contains exactly all vertices in the connected component of u .

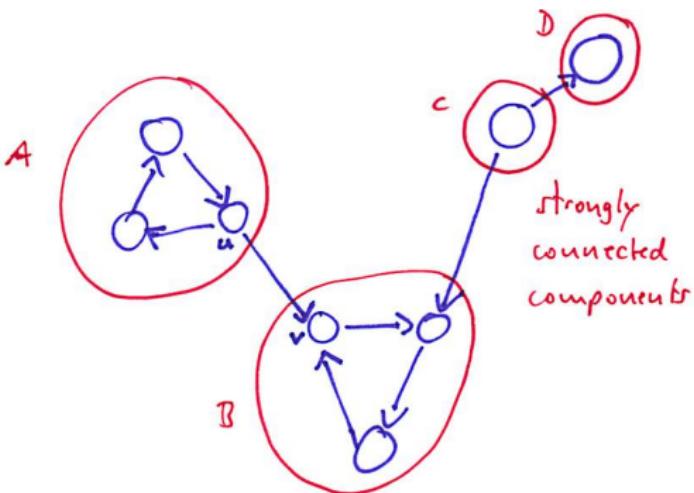
Consequence: each tree created by DFS exactly corresponds to a connected component of the graph.

Application: Strongly connected components

Literature: Cormen 22.5; Dasgupta

Component graph G^{SCC}

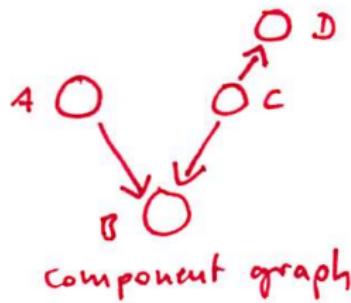
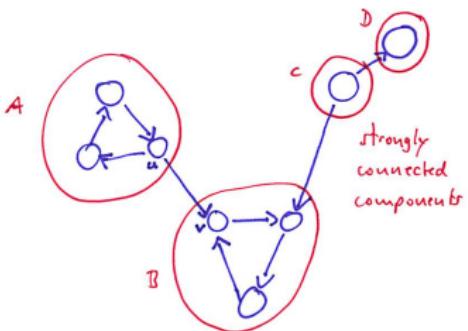
Recap: in a directed graph, a **strongly connected component** is a maximal subset S of vertices such that for each two vertices $u, v \in S$ there exists a directed path from u to v and vice versa.



Component graph G^{SCC} (2)

We define the component graph G^{SCC} of a directed graph:

- ▶ Vertices of G^{SCC} correspond to the components of G
- ▶ There exists an edge between vertices A and B in G^{SCC} if there exist vertices u and v in the connected components represented by A and B such that there is an edge from u to v .



Component graph G^{SCC} (3)

Proposition 3

For any directed graph G , the graph G^{SCC} is a DAG.

DO YOU SEE WHY THIS IS TRUE?

Component graph G^{SCC} (4)

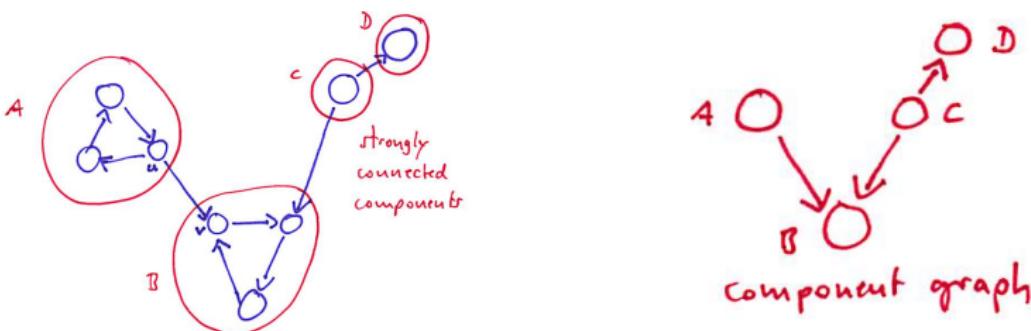
Proof: If G^{SCC} would contain a cycle, then all corresponding vertices would be part of a larger SCC, which contradicts the definition of the SCCs.



Component graph G^{SCC} (5)

We call a connected component a

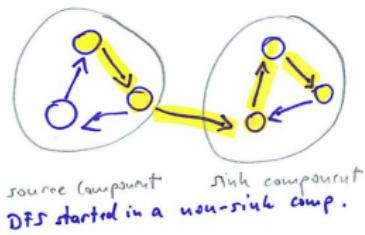
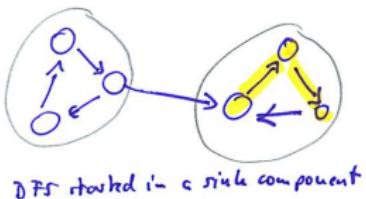
- ▶ **sink component** if the corresponding vertex in G^{SCC} does not have an out-edge (example: B and D in the figure)
- ▶ **source component** if the corresponding vertex in G^{SCC} does not have an in-edge (example: A and C in the figure)



Note: Any component is either sink or source

Observation: DFS when started in a sink comp.

- ▶ Consider a sink component B
- ▶ If we start DFS on G in a vertex $u \in B$, then the tree constructed in $\text{DFS}(G,u)$ covers the whole component B .
- ▶ However, if we start with u in a non-sink component, then the tree constructed in $\text{DFS}(G,u)$ covers more than this component.



- ▶ Idea: to discover the SCCs we start a DFS in a sink component and then work our way backwards along G^{SCC} .

Finding sources

Consider a standard DFS on a directed graph.

Define the **discovery time** $d(u)$ and **finishing time** $f(u)$ of a vertex as the time when the DFS algorithm first visits u (i.e., it makes u grey) and the time when it is done with u (i.e., it makes u black).

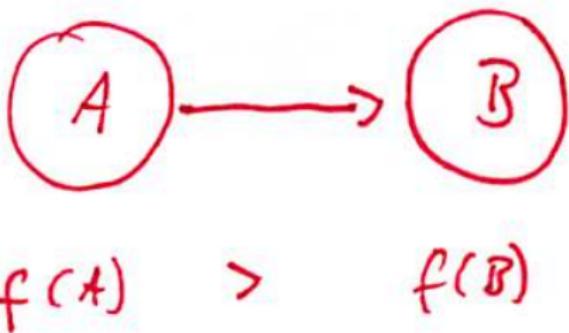
For a set A of vertices, define:

$$d(A) := \min_{u \in A} d(u) \quad \text{and} \quad f(A) := \max_{u \in A} f(u)$$

Finding sources (2)

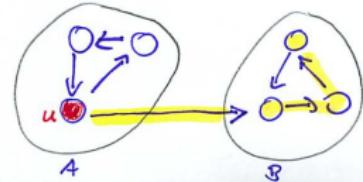
Proposition 4

Let A and B be two SCCs of a graph G and assume that B is a descendent of A in G^{SCC} . Then $f(B) < f(A)$ (no matter where we start the DFS).

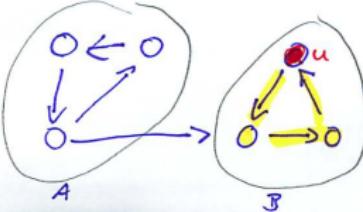


Finding sources (3)

Proof: Case $d(A) < d(B)$. Denote by u the first vertex we visit in A . Then the DFS builds a tree starting from u , and this tree will cover all of B before it is done with u .



Case $d(A) > d(B)$. Denote by u the first vertex we visit in B . Then the DFS will first visit all of B before moving on.



□

Finding sources (4)

Consequence:

- ▶ Assume we run a normal DFS on G and for each vertex we store the finishing time $f(u)$.
- ▶ By Proposition 4 we now know that the vertex u with the largest finishing time has to be in a source component of G^{SCC} .

Converting sources to sinks

By now we know how to find a source. But we would like to have a sink!

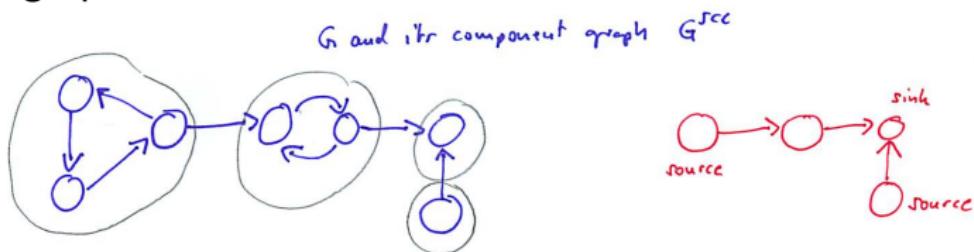
ANY IDEA?

Converting sources to sinks (2)

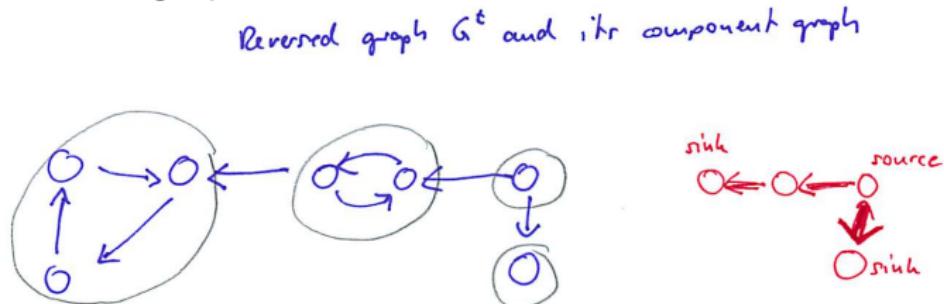
Trick: We “reverse” the graph: we consider the graph G^t which has the same vertices as G but all edges with reversed directions. Note that G^t has the same strongly connected components as G .

Converting sources to sinks (3)

Original graph:



Reversed graph:



SCC — final algorithm

General idea:

- ▶ Run a first DFS on G , with any arbitrary starting vertex. The vertex u^* with the largest finishing time $f(u)$ is in a source of G^{SCC} .
- ▶ Thus the vertex u^* is in a sink of $(G^t)^{SCC}$.
- ▶ Now start a second DFS on u^* in G^t . The tree discovered by $DFS(G^t, u^*)$ is the first strongly connected component.
- ▶ Then continue with a DFS on the vertex $V = v^*$ that has the highest $f(u)$ among the remaining vertices.
- ▶ And so on.

SCC — final algorithm (2)

- 1 $\text{SCC}(G)$
- 2 Call $\text{DFS}(G)$ to compute the finishing times $f(u)$
- 3 Compute the reverse graph G^t
- 4 Call $\text{DFS}(G^t)$, where the vertices in the main loop are considered in order of decreasing $f(u)$
- 5 Output the vertices of each tree of $\text{DFS}(G^t)$ as a SCC

Comments

Running time: as two DFSs, i.e. still $O(|V| + |E|)$.

Application: Topological sort

Literature: Cormen 22.5; Dasgupta Sec. 3.3.2; Mehlhorn Sec. 9.2.1

A scheduling problem to start with

A simple scheduling problem:

- ▶ You have to schedule a number of jobs.
- ▶ There are some dependencies like
 - ▶ job_3 can only happen after job_{13} is finished.
 - ▶ job_7 can only happen after job_1 is finished.
 - ▶ ...
- ▶ Your task:
 - ▶ Figure out whether a correct scheduling is possible at all.
 - ▶ Find a schedule (in what order do we perform the tasks).

ANY IDEAS HOW TO DO THIS?

A scheduling problem to start with (2)

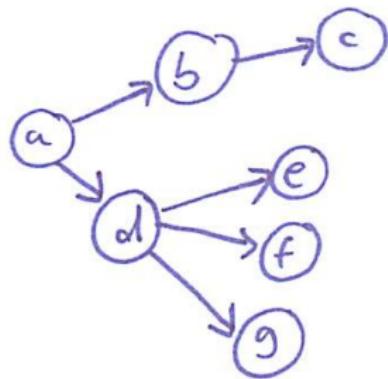
Solution:

- ▶ Form a directed graph from your constraints:
 - ▶ vertices correspond to jobs
 - ▶ put an edge from job_i to job_j if job_i has to be finished before job_j can start.
- ▶ Then try to find a “topological sort”:

Definiton:

A **topological sort** of a directed graph is a linear ordering of its vertices such that whenever there exists a directed edge from vertex u to vertex v , u comes before v in the ordering.

A scheduling problem to start with (3)



Topological sort:

a b c d e f g

First thoughts: when is it possible at all?

DOES A TOPOLOGICAL SORT EXIST FOR ANY DIRECTED
GRAPH?

First thoughts: when is it possible at all? (2)

No, only if the graph is a DAG. (WHY???)

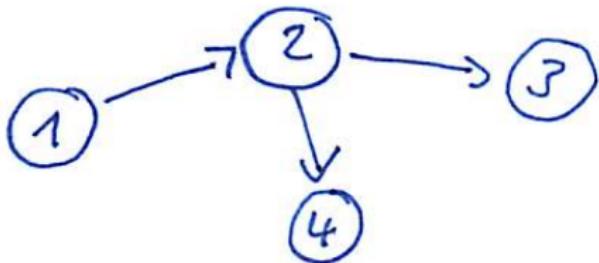
Exercise: prove it formally!

First thoughts: when is it possible at all? (3)

IN CASE THERE EXISTS A TOPOLOGICAL SORT, IS IT
ALWAYS UNIQUE?

First thoughts: when is it possible at all? (4)

No, in most cases not.



This graph is consistent with the orderings 1, 2, 3, 4 and 1, 2, 4, 3.

First thoughts: when is it possible at all? (5)

Proposition 5 (Uniqueness of topological sort)

The topological sort of a DAG is unique if and only if the DAG contains a Hamiltonian path (that is, a path that visits each vertex exactly once).

Proof: exercise.

Topological sort: how to solve it

Proposition 6 (DFS on a directed graphs)

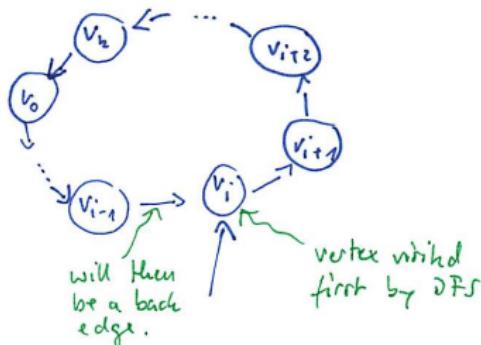
A directed graph has a cycle if and only if its DFS reveals a back edge (that is, an edge that goes from the current vertex to a previously visited vertex).

Proof of " \Leftarrow ":

Assume DFS reveals a back edge. This means that the DFS has already visited u , some intermediate vertices v_1, \dots, v_k , and then v , and now sees a link back to u . Thus, the graph has the cycle u, v_1, \dots, v_k, v, u .

Topological sort: how to solve it (2)

Proof of " \implies ":



- ▶ Assume there is a cycle $v_0, v_1, \dots, v_k, v_0$ in the graph.
- ▶ Let v_i be the vertex with the smallest discovery time.
- ▶ All other v_j can be reached from v_i and are thus descendants in the DFS tree.
- ▶ In particular, this is true for v_{i-1} , which is going to reveal the edge (v_{i-1}, v_i) , which is a back edge.

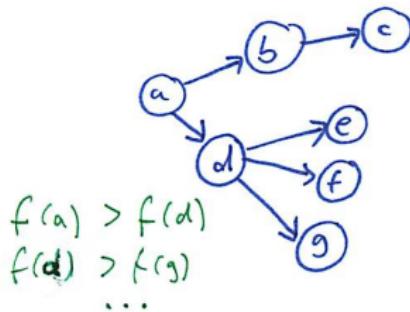


Topological sort: how to solve it (3)

Proposition 7

If we run DFS on a DAG, all graph edges go from larger to smaller finishing times.

- ▶ DFS on a DAG does not have any back edges
- ▶ Then we always finish the descendants first before we finish the ancestors.



Topological sort: how to solve it (4)

Make sure it is clear to yourself: Proposition 7 would not be true if there were back edges (find an example).

Topological sort: how to solve it (5)

Algorithm for topological sort:

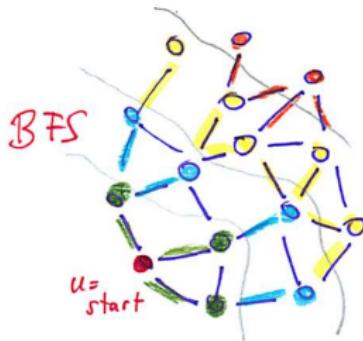
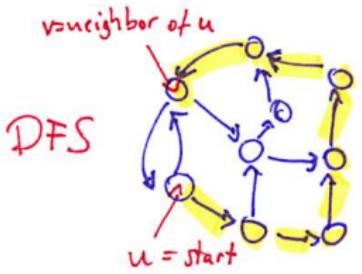
- ▶ Run DFS with an arbitrary starting vertex and record the finishing times.
- ▶ Then sort the vertices by decreasing finishing times.

Breadth first search

Literature: Cormen 22.2 (Dasgupta 4.2)

BFS — Intuition

In DFS, we can happen to take a very long path between two vertices that are very close indeed.



In BFS, we try to travel “uniformly” through the graph, making sure we first explore the local neighborhood of the starting point before going further.

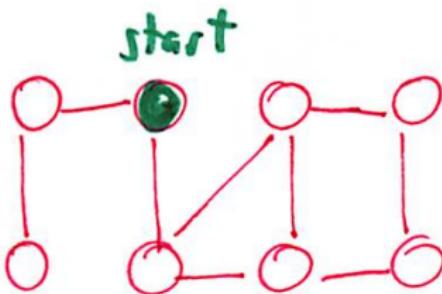
BFS — pseudocode

BFS(G, s)

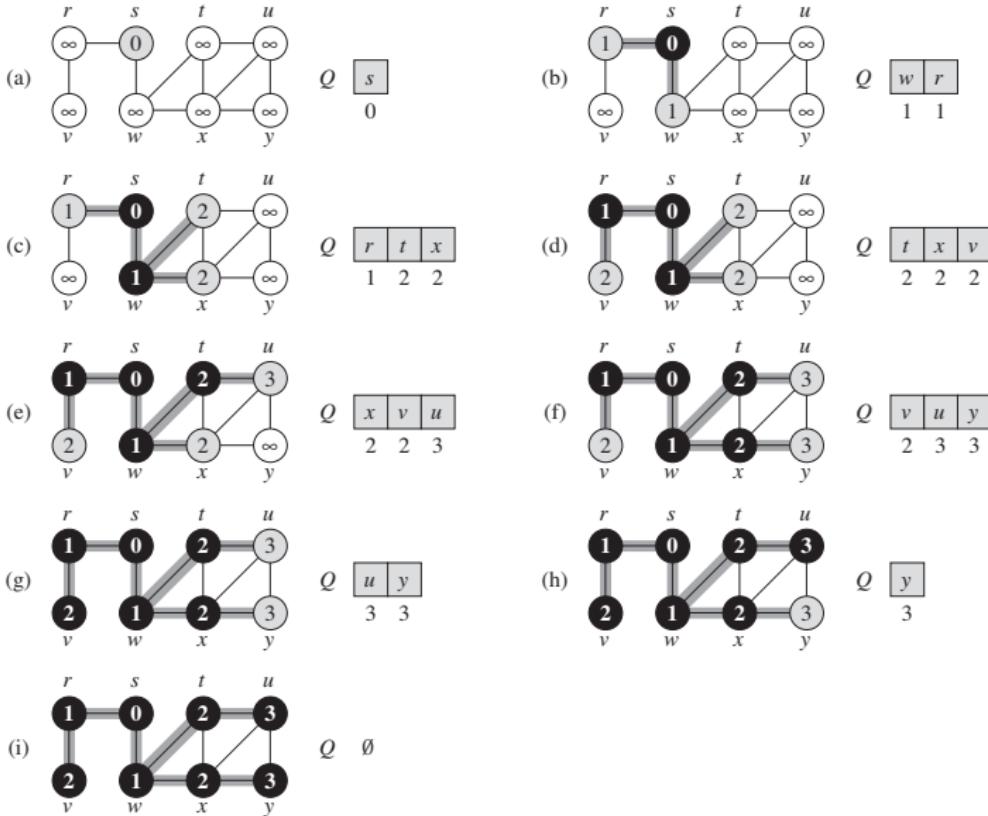
```
1 for all vertices  $u \in V \setminus \{s\}$  # initialize
2    $u.\text{color} = \text{white}$ 
3    $s.\text{color} = \text{grey}$ 
4    $Q = [s]$  # queue containing  $s$ 
5   while  $Q \neq \emptyset$ 
6      $u := \text{DEQUEUE}(Q)$ 
7     for all  $v \in \text{Adj}(u)$ 
8       if  $v.\text{color} == \text{white}$ 
9          $v.\text{color} = \text{grey}$ 
10         $\text{INQUEUE}(Q, v)$ 
11      $u.\text{color} = \text{black}$ 
```

BFS — an example

EVERYBODY SHOULD TRY TO RUN BFS ON THE FOLLOWING LITTLE EXAMPLE ON HIS OWN:



BFS — an example (2)



BFS — running time

- ▶ Running time: $O(|E| + |V|)$ (Why?)

BFS vs. DFS

- ▶ DFS travels “deep”, BFS travels more “like a wave”
- ▶ DFS is “adventurous”: it only traces back if it has nowhere else to go.
BFS is “timid”: it always traces back until it cannot trace back any more and has to move forward.
- ▶ On a high level: both are nearly the same, the main difference is that BFS uses a queue whereas DFS uses a stack.

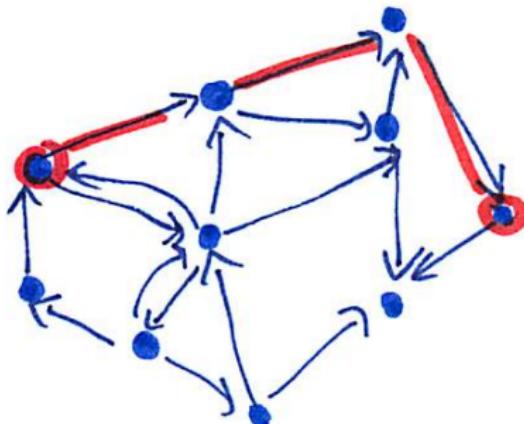
Application: shortest paths in unweighted graphs

Simple shortest path problem

In an unweighted graph, the length $\ell(\pi)$ of a path in the graph is defined as the number of edges in the path.

The shortest path distance between two vertices is defined as

$$d(u, v) = \min\{\ell(\pi) \mid \pi \text{ path between } u \text{ and } v\}.$$



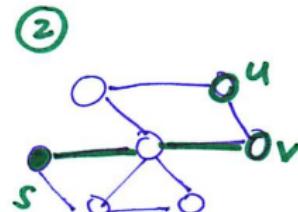
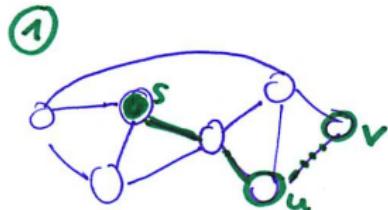
Simple shortest path problem (2)

Important property:

The shortest path distance satisfies the triangle inequality:

$$d(u, v) \leq d(u, w) + d(w, v) \text{ for all } u, v, w \in V$$

In particular: if there is an edge from u to v , then
 $d(s, v) \leq d(s, u) + 1$.



Simple shortest path problem (3)

Example:

- ▶ You want to find the shortest path in a network of roads.
- ▶ Assume all roads have the same lengths.

ANY IDEAS HOW THIS COULD BE SOLVED USING DFS OR BFS?

BFS finds shortest paths

Want to show: If G is an unweighted graph, then $BFS(G, s)$ can compute the shortest path distances of all vertices to s .

Let's insert a couple of lines in BFS:

BFS finds shortest paths (2)

BFS(G, s)

```
1 for all vertices  $u \in V \setminus \{s\}$ 
2    $u.\text{color} = \text{white}$ 
3    $u.\text{dist} = \infty$ 
4    $s.\text{dist} = 0$ 
5    $s.\text{color} = \text{grey}$ 
6    $Q = [s]$ 
7   while  $Q \neq \emptyset$ 
8      $u := \text{DEQUEUE}(Q)$ 
9     for all  $v \in \text{Adj}(u)$ 
10       if  $v.\text{color} == \text{white}$ 
11          $v.\text{color} = \text{grey}$ 
12          $v.\text{dist} = u.\text{dist} + 1$ 
13          $\text{INQUEUE}(Q, v)$ 
14        $u.\text{color} = \text{black}$ 
```

BFS finds shortest paths (3)

We are now going to prove the following statement:

Theorem 8

Let G be an unweighted graph. Upon termination of $BFS(G, s)$ we have $v.dist = d(s, v)$ for all vertices v that are reachable from s .

Even though it looks obvious, we are going to give a formal proof now. It requires several steps.

BFS finds shortest paths (4)

Proposition 9

Upon termination of $BFS(G, s)$, we have $v.dist \geq d(s, v)$ for all $v \in V$.

Proof:

We are going to prove the following **invariant**: **whenever a vertex is being enqueued, then it satisfies $v.dist \geq d(s, v)$.**

We use **induction** to prove this, where the induction runs over the number of INQUEUE operations.

Base case: s is put in the queue. Obvious because of initialization

BFS finds shortest paths (5)

Inductive hypothesis: We assume the statement is true for all vertices that have already been inserted in the queue (=grey and black vertices).

Inductive step: want to show that it is true for the next vertex that is going to be inserted to the queue.

- ▶ Consider a white vertex v that is discovered from predec. u .
- ▶ By the inductive hypothesis: $u.dist \geq d(s, u)$
- ▶ Then we get:

$$\begin{aligned} v.dist &= u.dist + 1 \text{ (by the pseudo-code)} \\ &\geq d(s, u) + 1 \text{ (by the inductive hypothesis)} \\ &\geq d(s, v) \text{ (by the triangle inequality)} \end{aligned}$$



BFS finds shortest paths (6)

Proposition 10

Suppose that during the execution of BFS the queue contains (v_1, \dots, v_r) where v_1 is the head and v_r the tail. Then:

$$v_1.dist \leq v_2.dist \leq \dots \leq v_r.dist \leq v_1.dist + 1$$

Proof: Induction on queue operations.

Base case: The queue only contains s . Obviously true.

Inductive step: Have to consider two cases, dequeue and enqueue operations.

BFS finds shortest paths (7)

- ▶ Assume we dequeue v_1 . Obviously we still have $v_2.dist \leq \dots \leq v_r.dist$. And we also have $v_r.dist \leq v_1.dist + 1 \leq v_2.dist + 1$ (using the inductive hypothesis $v_1.dist \leq v_2.dist$).
- ▶ Assume we enqueue v , it is now v_{r+1} .
Assume predecessor of v was u . u has just been dequeued.
Because u was previously head in the queue, by ind.hyp. we have $u.dist \leq v_1.dist$. By the shortest path property above we know $v.dist \leq u.dist + 1$. Together:
 $v.dist \leq u.dist + 1 \leq v_1.dist + 1$

Similarly can also conclude: $v_r.dist \leq u.dist + 1 = v.dist$



BFS finds shortest paths (8)

Finally, we are going to prove Theorem 8:

Proof: By Proposition 9 we know that for all vertices, $v.dist \geq d(s, v)$, that is BFS can only attain values that are too large.

Assume the theorem is wrong. Let v be vertex with the smallest $d(v, s)$ that gets a wrong distance. Let u be the predecessor vertex of v in a shortest path from s to v . In particular:
 $d(s, u) + 1 = d(s, v)$.

Consider the time when we dequeue u .

1. CASE $v.color = \text{white}$: then code sets $v.dist = u.dist + 1$, which would be the correct distance. ↳
2. CASE $v.color = \text{black}$: then v was ejected from the queue before u , contradicting Proposition 10. ↳

BFS finds shortest paths (9)

3. CASE $v.\text{color} = \text{grey}$: this happened for dequeuing another vertex w . Have by Proposition 10:
 $v.\text{dist} = w.\text{dist} + 1 \leq u.\text{dist} + 1$. ↴



Shortest path problems

Shortest paths: definitions and properties

We are now going to treat various shortest path algorithms more deeply.

- ▶ In an unweighted graph, the **length $\ell(\pi)$ of a path** in the graph is defined as the number of edges in the path. In a weighted graph, the length of a path is usually defined as the sum of the edge weights along the path.
- ▶ The **shortest path distance** between two vertices is defined as

$$d(u, v) = \min\{\ell(\pi) \mid \pi \text{ path between } u \text{ and } v\}.$$

- ▶ Observe: shortest paths are often not unique.
- ▶ Attention: in the presence of negative weights, funny things can happen. In particular, if the graph contains loops in which the sum of weights is negative, then the shortest path has length $-\infty$ (hence it is not well-defined).

Shortest paths: definitions and properties (2)

Proposition 11

The shortest path distance satisfies the triangle inequality:

$$d(u, v) \leq d(u, w) + d(w, v) \text{ for all } u, v, w \in V.$$

Proof: Easy.

Consequence: In graphs with non-negative weights, the shortest path distance is a **metric**

- ▶ $d(a, b) \geq 0$ for all a, b
- ▶ $d(a, b) = 0 \iff a = b$
- ▶ $d(a, b) \leq d(a, c) + d(c, b)$

Shortest paths: definitions and properties (3)

Proposition 12

Subpaths of shortest paths are shortest paths.

Proof: easy.

Different shortest path problems

We are going to consider several types of problems:

- ▶ Single Source Shortest Paths: We want to know the shortest path distances of one particular vertex s to all other vertices.
- ▶ All Pairs Shortest Paths: We want to know the shortest path distance between all pairs of points.
- ▶ Point to Point Shortest Paths: We want to the shortest path distance between a particular source vertex s and a particular sink vertex t

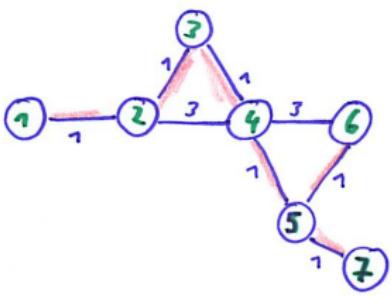
Storing paths efficiently

Keep track of the predecessors in the shortest paths with the help of the **predecessor matrix** $\Pi = (\pi_{ij})_{i,j=1,\dots,n}$:

- ▶ If $i = j$ or there is no path from i to j , set $\pi_{ij} = NIL$
- ▶ Else set π_{ij} as the predecessor of j on a shortest path from i to j .

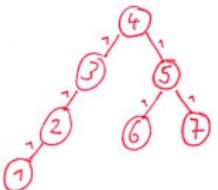
The subgraph induced by row i of the matrix Π induces a tree structure called the **predecessor subgraph** of G for i .

Storing paths efficiently (2)



	1	2	3	4	5	6	7
von	N	1	2	3	4	5	7
1	2	N	2	3	4	5	5
2	3	2	N	3	4	5	7
3	4	2	3	4	N	4	7
4	5	2	3	4	5	N	5
5	6	2	3	4	5	5	N
6	7	2	3	4	5	5	N
7							

predecessor subgraph of ④ =
tree corresponding to root ④:

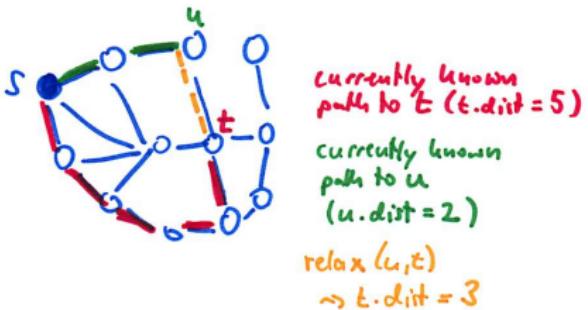


Single Source Shortest Paths

A general technique: Relaxation

... is going to be used by several of the following algorithms.

- ▶ For each vertex, keep an attribute $v.dist$ that is the current estimate of the shortest path distance to the source vertex s .
- ▶ Initially, it is set to ∞ for all vertices except the start vertex s .
- ▶ Relaxation step: figure out whether we can improve the shortest path from s to v by using an edge (u, v) and thus extending the shortest path of s to u .



A general technique: Relaxation (2)

This kind of procedure has several properties that are easy to prove:

- ▶ Upper bound: $v.dist$ is always an upper bound to the actual shortest path distance.
- ▶ No path property: If there is no path from s to v , then $v.dist = \infty$.
- ▶ Convergence: If $u.dist$ contains the correct shortest path distance, this never changes any more during the course of the algorithm.

A general technique: Relaxation (3)

Two functions we will need over and over again. Let G be a graph with weights w and s the starting point for our single source shortest path problem.

InitializeSingleSource(G, s)

```
1 for all  $v \in V$ 
2    $v.dist = \infty$  # Current distance estimate
3    $v.\pi = NIL$  # Predecessor on the best current path to  $v$ 
 $s.dist = 0;$ 
```

Relax(u, v)

```
1 if  $v.dist > u.dist + w(u, v)$ 
2    $v.dist = u.dist + w(u, v)$ 
3    $v.\pi = u$ 
```

Bellman-Ford algorithm

Literature: Cormen 24.1; (Kleinberg 6.8)

Original sources are somewhat hard to track, commonly cited are:

- ▶ R. Bellman: On a routing problem. Quarterly of Applied Mathematics 16: 87–90, 1958
- ▶ L. R. Ford: Network flow theory, Paper P-923. The Rand Corporation, Santa Monica 1956
- ▶ E. F. Moore: The shortest path through a maze. In: Proceedings of the International Symposium on the Theory of Switching. 2/1959. Harvard University Press, S. 285–292

See also the notes by D. Walden (2003) on the history of the Bellman-Ford algorithm (is in the paper repository)

Bellman-Ford algorithm: pseudo-code

Solves the single source shortest path problem for general weighted graphs (edges are allowed to be negative). Returns false in case of negative-weight-cycles.

BellmanFord(G, s)

```
1 InitializeSingleSource( $G, s$ )
2 for  $i = 1, \dots, |V| - 1$ 
3   for all edges  $(u, v) \in E$ 
4     Relax( $u, v$ )
5   for all edges  $(u, v) \in E$ 
6     if  $v.dist > u.dist + w(u, v)$ 
7       return false
8 return true
```

Analysis of the algorithm

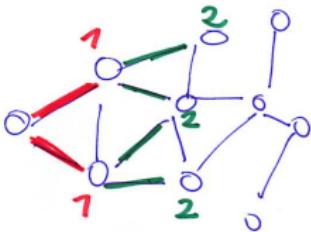
Proposition 13

Assume that G contains no negative weight cycles. Then the Bellman-Ford algorithm returns the correct shortest path values.

Proof : (intuition, case unweighted graphs)

In the first iteration of the for-loop, the edges to the neighbors of s are relaxed, the the neighbors of s have the correct values. After the second iteration, their neighbors have the correct values. And so on.

Analysis of the algorithm (2)



correct after iteration 1
correct after iteration 2

Analysis of the algorithm (3)

Formal proof:

Consider any vertex v in the graph, and a shortest path s, v_1, \dots, v_k, v from s to v .

- ▶ After the first iteration, v_1 has the correct distance.
- ▶ After the second iteration, v_2 has the correct distance.
- ▶ In general, after iteration i , all vertices whose shortest paths contain at most i edges have the correct distances.
 - ▶ To be very formal, use a proof by induction for this statement.
 - ▶ Note that this also holds for weighted graphs with arbitrary edge weights, as long as there is no negative cycle.
- ▶ Shortest paths in the graph can contain at most $|V| - 1$ edges. So after the last iteration, each vertex in the graph has the correct distance.



Analysis of the algorithm (4)

Proposition 14

If G contains a negative-weight cycle, then the algorithm returns FALSE.

Proof: Denote the negative-weight cycle by v_0, \dots, v_k with $v_0 = v_k$. By definition

$$\sum_{i=0}^{k-1} w(v_i, v_{i+1}) < 0.$$

Now assume the algorithm returns TRUE. In this case, the IF-clause at the end of the code was false for all v_i , that is

$$v_i.dist \leq v_{i-1}.dist + w(v_{i-1}, v_i)$$

Summing up these inequalities over $i = 0, \dots, k - 1$ and exploiting that $v_0 = v_k$ leads to $0 \leq \sum_{i=0}^{k-1} w(v_i, v_{i+1})$. \lhd



Running time

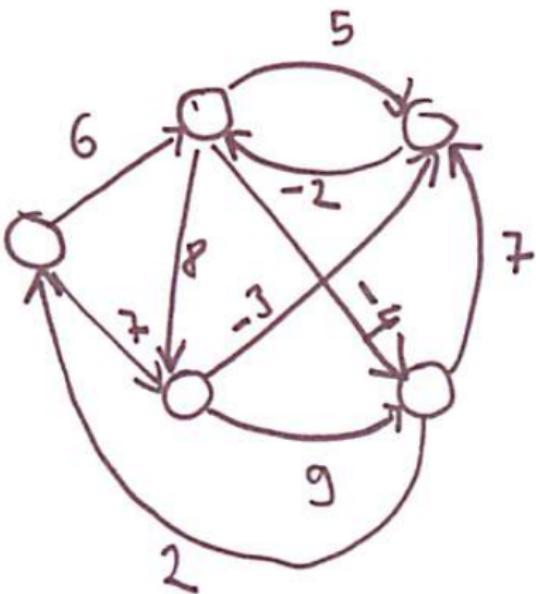
Proposition 15

The running time of the algorithm is $O(|V| \cdot |E|)$.

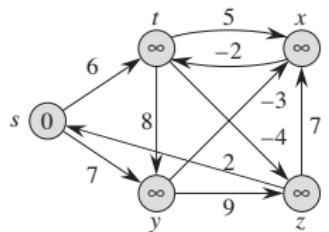
Proof: Easy

Example

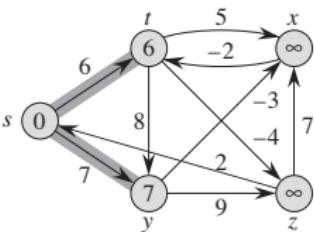
EVERYBODY: SOLVE THE FOLLOWING LITTLE EXAMPLE:



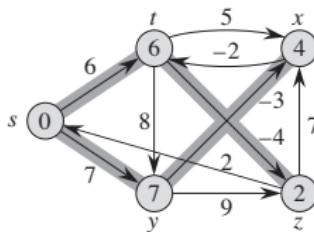
Example (2)



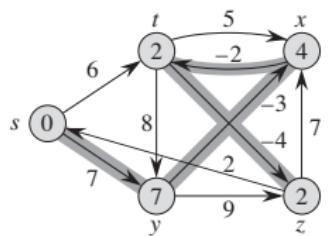
(a)



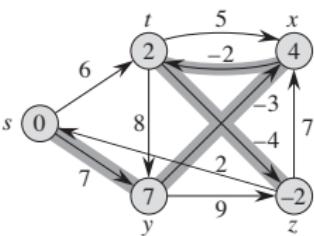
(b)



(c)



(d)



(e)

Assumptions

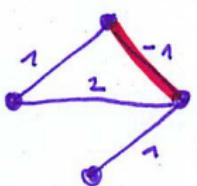
Note: The Bellman-Ford algorithm works on all general kinds of graphs: weighted or unweighted, directed or undirected, weights arbitrary.

However, note the following for undirected graphs:

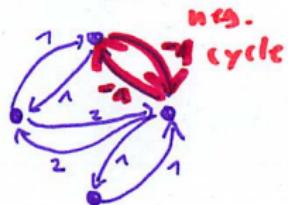
- ▶ If you run the Bellman-Ford algorithm on an undirected graph, you always have to relax the edges in both directions. The easiest is to convert the undirected graph in the equivalent directed graph.
- ▶ In particular, if there is an edge with a negative weight in an undirected graph, this induces a negative weight cycle in the directed version!!! So whenever an undirected graph has negative weights, shortest path is not defined.

Assumptions (2)

Undirected graph



Corresponding directed graph



Underlying paradigm: dynamic programming

The Bellman-Ford algorithm is an example of **Dynamic Programming**.

- ▶ decompose problem into smaller subproblems
- ▶ build up the overall solution by larger and larger subproblems (which are allowed to overlap, as opposed to divide and conquer)
- ▶ Store the solutions to all the subproblems solved so far in a table

In this particular case the crucial observation is as follows:

$$\ell(\text{shortest path of length } i \text{ from } s \text{ to } t)$$

$$= \min_u \ell(\text{shortest path of length } i - 1 \text{ from } s \text{ to } u) + w(u, t)$$

Dijkstra's algorithm

Literature: Cormen 24.3; Kleinberg 4.4; Dasgupta 4.4.

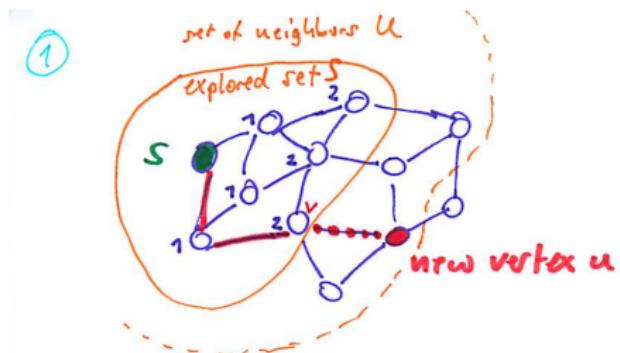
Original publication:

E. Dijkstra: A note on two problems in connexion with graphs.
Numerische Mathematik 1: 269–271, 1959

Dijkstra's algorithm: the naive way

This algorithm works on any weighted, directed (or undirected) graph **in which all edge weights $w(u, v)$ are non-negative**.

- ▶ Algorithm maintains a set S of vertices for which it already knows the shortest path distances from s .
- ▶ Then it looks at neighbors u of S and assigns a guess for the shortest path by using a path through S and adding one edge.



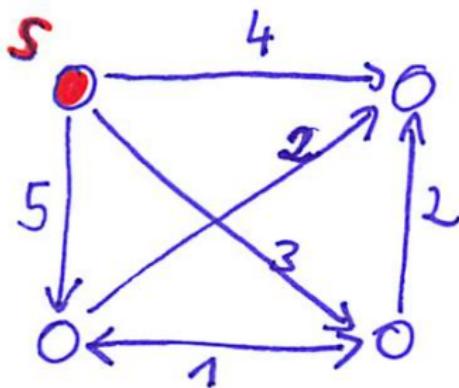
Dijkstra's algorithm: the naive way (2)

DijkstraNaive(G, w, s)

```
1  $S = \{s\}$  #  $S$  set of explored vertices
2  $d(s) = 0$ 
3 while  $S \neq V$ 
4    $U := \{u \notin S \mid u \text{ neigh. of a vertex } \in S\}$  # candidates
5   for all  $u \in U$ 
6     for all  $\text{pre}(u) \in S$  that are predecessors of  $u$ 
7        $d'(u, \text{pre}(u)) := d(\text{pre}(u)) + w(\text{pre}(u), u)$ 
         # candidate distances
8      $u^* := \operatorname{argmin}\{d'(u, \text{pre}(u)) \mid u \in U\}$  # choose best candidate
9      $d(u^*) = d'(u^*)$ 
10     $S = S \cup \{u^*\}$ 
```

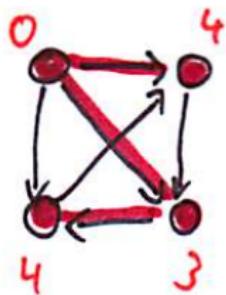
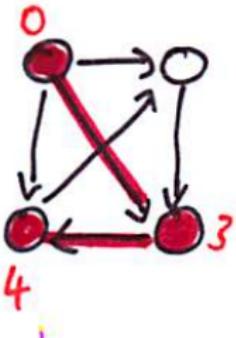
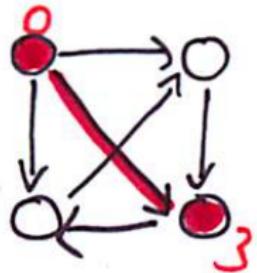
Examples

EVERYBODY, PLEASE TRY THIS EXAMPLE:



Examples (2)

Solution:



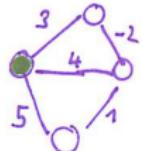
Examples (3)

Important: Dijkstra's algorithm can go wrong if a graph has negative-weight edges!

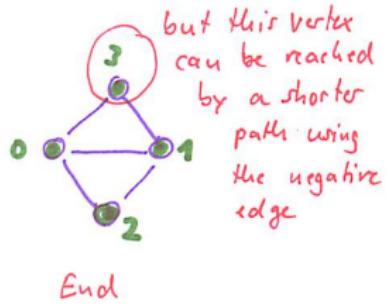
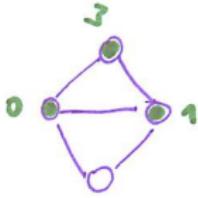
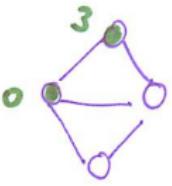
CAN YOU FIND AN EXAMPLE?

Examples (4)

Example where Dijkstra gives the wrong result:



Start



End

Analysis

Theorem 16 ((Correctness of the algorithm))

Consider the set S at any point in the algorithm. For each $t \in S$, the value $d(t)$ coincides with the shortest path distance from s to t .

Proof. Induction on the size $|S|$ of S .

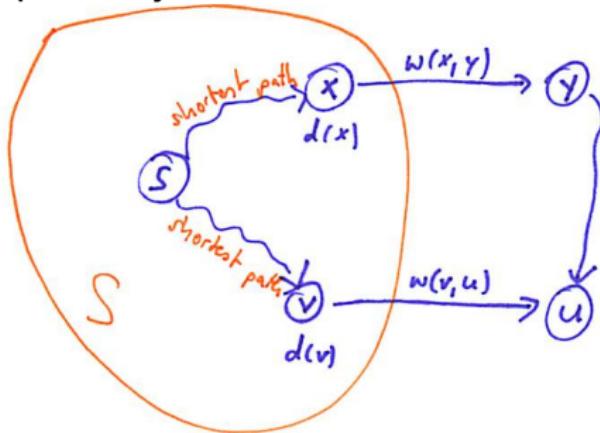
Base case: $|S| = 1$. Then $S = \{s\}$, and we assigned $d(s) = 0$. ☺

Inductive step from k to $k + 1$:

- ▶ Let u be the next vertex we add to S , denote its predecessor in S by v .
- ▶ Assume $d(u) = d(v) + w(v, u)$ is not the shortest path distance.
- ▶ Instead, consider a correct shortest path to u .

Analysis (2)

- It has to leave S somewhere, say at x , and say the next vertex on the shortest path is y .



- The path via x and y can only be shorter than the one via v if $w(x, y) < w(v, u)$.
- But as we added u and not y to S in this step, $w(x, y) \geq w(v, u)$ according to the algorithm. ↵

Analysis (3)



Running time of the naive implementation

Running time: the naive implementation DijkstraNaive has running time $O(|V| \cdot |E|)$:

- ▶ while loop runs for all $|V|$ vertices
- ▶ inside the while loop, we have to look at all neighbors of S , that is we have to look at all edges from S to $V \setminus S$, which is bounded by $|E|$ edges.

This is pretty bad!

Luckily we can save a lot by using an appropriate data structure.

Recap: min-priority queue

- ▶ Maintains a set S of elements
- ▶ Each element s has a key $k(s)$ assigned to it
- ▶ The key of each element can be *decreased* at any point.
- ▶ When extracting an element, we extract the one with the smallest key.

Implementations: using arrays or various heaps, see later.

Implementation	extract element	insert element / decrease key
Array	$O(V)$	$O(1)$
Binary heap	$O(\log V)$	$O(\log V)$
d-ary hep	$O\left(\frac{d \log V }{\log d}\right)$	$O\left(\frac{\log V }{\log d}\right)$
Fibonacci heap	$O(\log V)$	$O(1)$ (amortized)

Literature: Cormen 6.5; Kleinberg 2.5

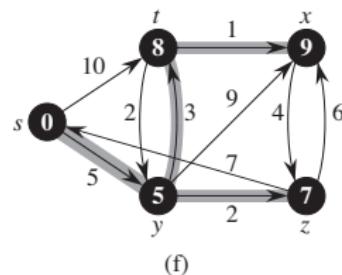
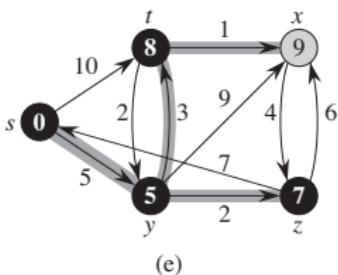
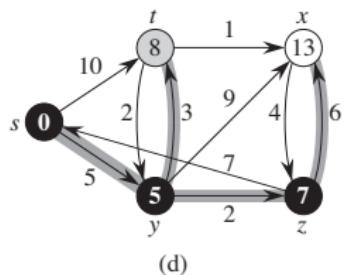
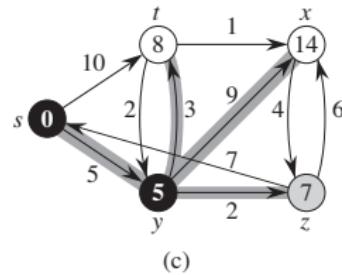
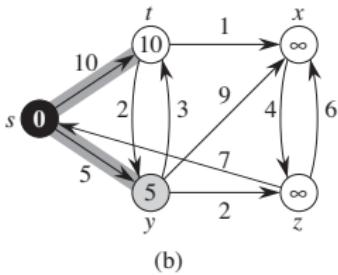
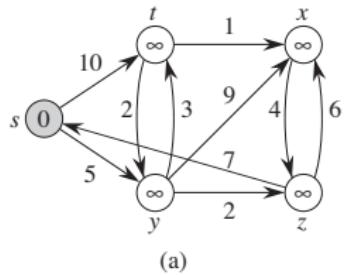
Dijkstra's algorithm with min-priority queues

Dijkstra(G, w, s)

- 1 InitializeSingleSource(G, s) *# initializes the $v.dist$ values*
- 2 $Q = (V, V.dist)$ *# insert all vertices with the initial keys $v.dist$*
- 3 **while** $Q \neq \emptyset$
- 4 $u = \text{Extract}(Q)$
- 5 **for all** v adjacent to u
- 6 Relax(u, v) and update the keys in Q accordingly

Note: Intuitively, Q is the complement of the set S we had in the naive Dijkstra algorithm: $Q = V \setminus S$.

Dijkstra's algorithm with min-priority queues (2)



Dijkstra's algorithm with min-priority queues (3)

Exercise: Formally prove that the two algorithm based on priority queues does exactly the same as the naive Dijkstra algorithm.

Dijkstra's algorithm with min-priority queues (4)

Running time:

- ▶ $|V|$ Insert operations of the queue
- ▶ $|V|$ Extract operations of the queue
- ▶ $|E|$ DecreaseKey operations of the queue (in the Relax subfunction)

Depending on how we implement the min-priority queue we get the following overall running times for Dijkstra's algorithm:

- ▶ Implementation by a naive array $\sim O(V^2)$
- ▶ Implementation by a binary heap $\sim O((V + E) \log V)$
- ▶ Implementation by a d -ary heap $\sim O(|V| \cdot d + |E|) \frac{\log V}{\log d}$);
- ▶ Implementation by a Fibonacci heap $\sim O(V \log V + E)$

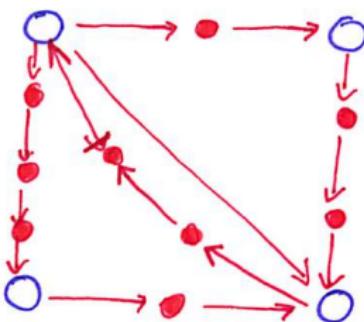
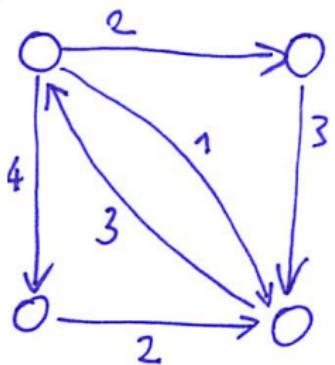
Which one to choose?

Dijkstra's algorithm with min-priority queues (5)

- ▶ If the graph is **dense**, that is $|E| = O(|V|^2)$: prefer the array implementation
- ▶ If the graph is **sparse**, that is $|E|$ is (much) smaller than $|V|^2$: prefer a heap implementation
- ▶ For a d-ary heap: choose d depending on graph: $d = |E|/|V|$. Then:
 - ▶ If graph is very sparse, i.e. $|E| = \Omega(|V|)$ we get $O(|V| \log |V|)$
 - ▶ If graph is dense, i.e. $|E| = \Omega(|V|^2)$, we get $O(|V|^2)$

Dijkstra's algorithm and BFS

One can interpret Dijkstra's algorithm as a generalization of BFS.
For simplicity assume that edge weights are integers. Replace all edges of length k by a string of $k - 1$ vertices connected by edges of length 1:



Then BFS on the new graph does the same as Dijkstra on the old graph.

Remarks

- ▶ In some sense, Dijkstra's algorithm is a **greedy** algorithm. At each point in time, it does the “locally best” best action. In this particular case, the solution also turns out to be “globally optimal”.
- ▶ Dijkstra's algorithm requires global knowledge of the network (maintain set S ; make global decision about which node to add next). Unsuitable for many applications where global network is unknown or too big (e.g., routing problems in the www).

Generic labeling method

Generic labeling method

A convenient generalization of Dijkstra and Bellman-Ford:

- ▶ For each vertex, maintain a status variable $S(v) \in \{\text{unreached}, \text{labelchanged}, \text{settled}\}$.
- ▶ Repeatedly relax edges.
- ▶ Do this until nothing changes any more.

Pseudo-Code is as follows:

Generic labeling method (2)

GenericLabelingMethod(G, s)

```
1 for all  $v \in V$ 
2    $v.dist = \infty$  # current distance estimate
3    $v.parent = NIL$  # parent in current shortest path
4    $v.status = \text{unreached}$ 
5    $s.dist = 0$ 
6    $s.status = \text{labelchanged}$ 
7   while there exist vertices with status "labelchanged"
8     pick one such vertex  $v$ 
9     for all neighbors  $u$  of  $v$ 
10       Relax( $v, u$ )
11       if this relaxation changed the value  $u.dist$ 
12          $u.status = \text{labelchanged}$ 
13        $v.status = \text{settled}$ 
```

Generic labeling method (3)

- ▶ Note that it can happen that a vertex gets scanned several times, the status can change back from “settled” to “labelchanged”.
- ▶ How often this is going to happen (and thus, how efficient this generic method is) depends on how exactly we choose the next vertex in line 8 of the code.
- ▶ If we manage to always select a vertex v such that $v.dist$ is already the correct distance, then each vertex gets scanned exactly once.
- ▶ Dijkstra’s algorithm is a special case of the labeling method: one can show that if we choose the vertex that has the smallest $v.dist$ value among all vertices with status “labelchanged”, then the algorithm is equivalent to Dijkstra and each vertex gets scanned at most once.

Generic labeling method (4)

- ▶ The Bellman-Ford algorithm can also be written in this framework (see later: decentralized Bellman-Ford)
- ▶ Another example we will see later: A^* search

All pairs shortest paths on general graphs

All pairs shortest paths

Goal: want to find the shortest paths between *all* pairs of vertices in a graph.

Naive approach: run Bellman-Ford or Dijkstra with all possible start vertices

Problem with this approach:

- ▶ Running time!
 - ▶ Bellman-Ford $\sim O(V^2 \cdot E)$
 - ▶ Dijkstra, say with binary heaps $\sim O(V(V + E) \log V)$
- ▶ Seems like a waste of efforts that we do not “re-use” the results we already have

Floyd-Warshall algorithm

Literature: Cormen 25

Original papers:

- R. Floyd. Algorithm 97: Shortest Path. Communications of the ACM 5 (6), 1962
- S. Warshall. A Theorem on Boolean Matrices. J. ACM 9, 11-12, 1962.

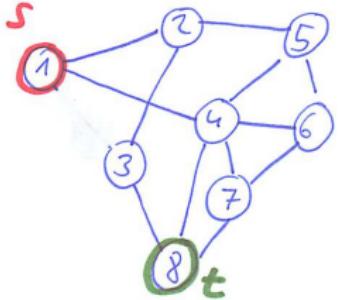
Floyd-Warshall algorithm — idea

Dynamic programming approach:

- ▶ Assume all vertices are numbered from 1 to n .
- ▶ Fix two vertices s and t .
- ▶ Consider all paths from s to t that only use vertices $1, \dots, k$ as intermediate vertices. Let $\pi_k(s, t)$ be a shortest path *from this set*, and denote its length by $d_k(s, t)$.
- ▶ Want to find a recursive relation between the π_k and π_{k-1} to be able to construct the solution bottom-up.

Floyd-Warshall algorithm — idea (2)

Example for the definition of π_k and d_k :



$$\begin{array}{lll}
 \pi_0(s, t) = \text{NIL} & d_0(s, t) = \infty \\
 \pi_1(s, t) = \text{NIL} & d_1(s, t) = \infty \\
 \pi_2(s, t) = \text{NIL} & d_2(s, t) = \infty \\
 \pi_3(s, t) = [1\ 2\ 3\ \delta] & d_3(s, t) = 3 \\
 \pi_4(s, t) = [1\ 4\ \varnothing] & d_4(s, t) = 2
 \end{array}$$

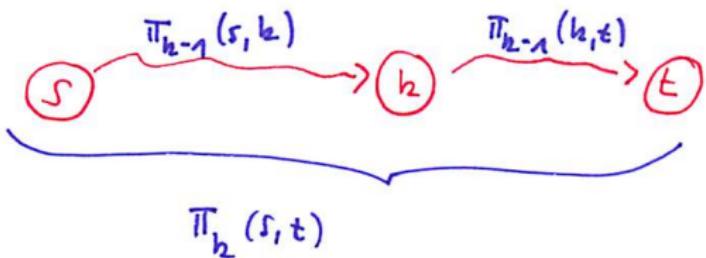
Floyd-Warshall algorithm — idea (3)

Observe the following recursive relationships:

- ▶ If vertex k is not element of $\pi_k(s, t)$, then
 $\pi_k(s, t) = \pi_{k-1}(s, t).$
- ▶ If vertex k is an element of π_k , it occurs at most once. Thus we can decompose

$$\underbrace{[sv_1 \dots v_i k v_{i+1} \dots v_j t]}_{\pi_k(s, t)} = \underbrace{[sv_1 \dots v_i k]}_{\pi_{k-1}(s, k)} \circ \underbrace{[kv_{i+1} \dots v_j t]}_{\pi_{k-1}(k, t)}$$

Floyd-Warshall algorithm — idea (4)



Floyd-Warshall algorithm — idea (5)

To exploit this insight bottom-up, we proceed as follows:

Case $k = 0$ (“no intermediate vertices”): Define

$$d_0(s, t) = w(s, t)$$

Case $k > 0$: recursively define

$$d_k(s, t) = \min\{d_{k-1}(s, t) , d_{k-1}(s, k) + d_{k-1}(k, t)\}$$

In the algorithm below, we define the $n \times n$ matrix D_k as the matrix with entries $(d_k(s, t))_{s,t=1,\dots,n}$.

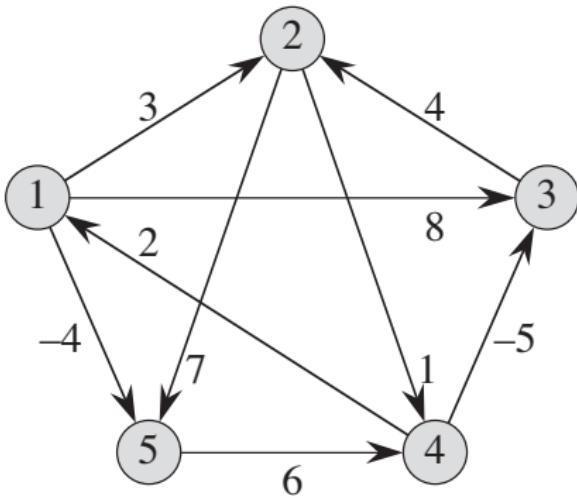
Floyd-Warshall — pseudo code

FloydWarshall(W)

```
1  $n :=$  number of vertices
2  $D^{(0)} = W$  # Matrix of size  $n \times n$ 
3 for  $k = 1, \dots, n$ 
4   Let  $D^{(k)}$  be a new  $n \times n$  matrix
5   for  $s = 1, \dots, n$ 
6     for  $t = 1, \dots, n$ 
7        $d_k(s, t) = \min\{d_{k-1}(s, t), d_{k-1}(s, k) + d_{k-1}(k, t)\}$ 
8       #  $d_k(s, t)$  is the entry at  $(s, t)$  in Matrix  $D^{(k)}$ 
8 return  $D^{(n)}$ 
```

Example

Consider this graph:



On the next slide we see the corresponding $D^{(k)}$ -matrices:

Example (2)

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ 8 & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Floyd-Warshall — running time

Running time: $O(|V|^3)$.

- ▶ Hm, still does not look too good 😞
- ▶ Seems like it saves surprisingly little compared to other algorithms, even though it “re-uses” knowledge.

As a comparison: Using Dijkstra, say with binary heaps, leads to $O(V(V + E) \log V)$.

- ▶ If the graph is sparse, then Dijkstra is obviously preferable.
- ▶ If the graph is dense, Floyd-Warshall is preferable.

In general, Floyd-Warshall is simpler to implement and works reasonably on moderately sized graphs.

And what if the graph has a negative-weight cycle?

We can detect this with Floyd-Warshall as follows:

- ▶ We have a negative cycle if there exists a path from i to i with negative length.
- ▶ Simply look at the values of the diagonal of the distance matrix that is output by Floyd-Warshall. If it contains negative entries, then the graph contains a negative cycle.
- ▶ Then all the results that were computed by the algorithm are meaningless.

Johnson's algorithm

Literature: Cormen 25.3

Original paper:

D. Johnson. Efficient algorithms for shortest paths in sparse networks. Journal of the ACM 24 (1): 1-13, 1977

Johnson's algorithm

Assume our graph has negative edges.

Naive idea: add weight to all edges such that all edge weights become positive: For some large constant a define

$$w'_{ij} = w_{ij} + a.$$

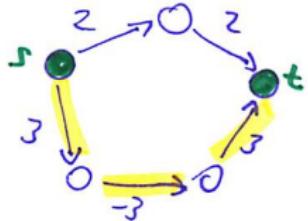
Then all weights are non-negative, and we can run Dijkstra without problems.

WHAT IS YOUR GUESS, IS THIS APPROACH VALID OR NOT?

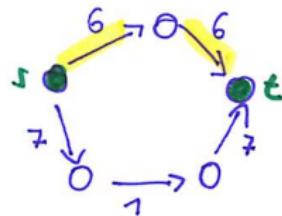
Johnson's algorithm (2)

⚠️Attention⚠️: this naive approach can change the shortest paths.

Example:



shortest path in orig. graph
(negative weights!)



shortest path after adding 4 to
all edges

Johnson's algorithm (3)

Instead we want:

Transform the edge weights such that

- (i) shortest paths don't change
- (ii) new edge weights are non-negative

If we manage to do this, we can proceed as follows:

- ▶ Given graph with arbitrary weights w_{ij}
- ▶ Transform its weights to new weights \tilde{w}_{ij} that satisfy (i) and (ii)
- ▶ Then find the shortest paths according to the new edge weights

Johnson's algorithm (4)

First insight, deals with (i):

Define new edge weights \tilde{w}_{ij} by

$$\tilde{w}_{ij} = w_{ij} + h(i) - h(j)$$

for some function $h : V \rightarrow \mathbb{R}$.

Theorem 17

Johnson's algorithm (5)

Given a weighted graph $G = (V, E)$ with arbitrary weight matrix W . Let $h : V \rightarrow \mathbb{R}$ be any function. Define the new edge weight matrix $\tilde{W} = (\tilde{w}_{ij})$ as above, and let \tilde{G} be the graph with the new edge weights.

1. Then π is a shortest path from s to t in G if and only if π is a shortest path from s to t in \tilde{G} .
2. G has a negative-weight cycle if and only if \tilde{G} has a negative weight cycle.

Johnson's algorithm (6)

Proof:

- ▶ Fix s and t . For any path π between s and t we have that $\tilde{\ell}(\pi) = \ell(\pi) + h(s) - h(t)$
- ▶ Thus shortest paths between a fixed pair s and t of vertices don't change.
- ▶ In particular, for any cycle π we have $\tilde{\ell}(\pi) = \ell(\pi)$ (just consider the case $s = t$ above).



Johnson's algorithm (7)

But how to choose the function h to satisfy (ii)?

If the graph has negative weight cycles:

- ▶ No matter what we choose as function h , we will always still have negative weight edges (because we still have negative weight cycles)!
- ▶ So we cannot get rid of negative cycles.
Note that this is a good feature!

If we don't have negative weight cycles:

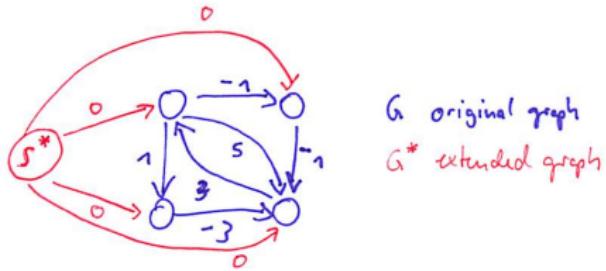
- ▶ $h(u)$ interferes with *all* neighbors of u .
- ▶ So we might start to turn one edge positive by assigning appropriate values to its end points u and v , but then these values interfere with all other neighbors of u and v as well...
- ▶ ... not obvious at all ...

Johnson's algorithm (8)

Here is the solution how to deal with (ii):

First, we add a new vertex s^* to the graph as follows:

- ▶ Add a directed edge from s^* to all vertices v with edge weight $w(s^*, v) := 0$.
- ▶ This does not generate new cycles in the graph.
- ▶ All shortest paths in the old graph remain shortest paths in the new graph (because the in-degree of s^* is 0).
- ▶ Let's call the new graph G^* .



Johnson's algorithm (9)

Now run the Bellman-Ford algorithm on G^* with source s^* :

- ▶ If the algorithm returns FALSE, then the graph G^* (and thus, G) contains a negative weight cycle. Stop.
- ▶ Otherwise, let $\delta(s^*, v)$ be the shortest path values for source s^* returned by Bellman-Ford. Note that $\delta(s^*, v) \leq 0$ for all $v \in V$ (why?).
- ▶ Now define $h(v) := \delta(s^*, v)$.

Johnson's algorithm (10)

Proposition 18

If the graph G does not contain any negative-weight cycle, then the edge weights \tilde{w}_{ij} generated with the function h from above are all non-negative.

Proof: By the triangle inequality, we have

$$h(v) := \delta(s^*, v) \leq \delta(s^*, u) + w(u, v) = h(u) + w(u, v),$$

that is

$$w(u, v) + h(u) - h(v) \geq 0.$$

Johnson's algorithm (11)

Thus $\tilde{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$.

□

All in all, this leads to the following algorithm:

Johnson's algorithm (12)

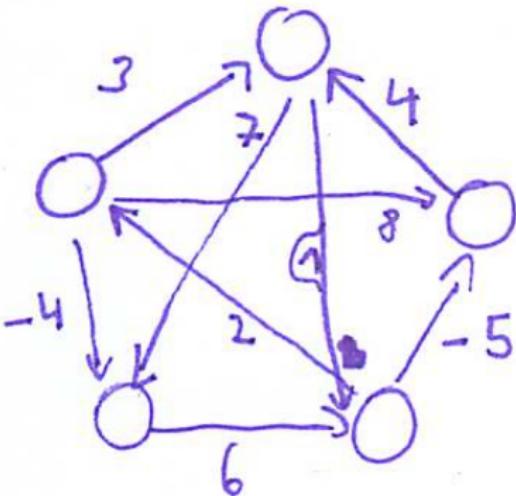
JohnsonAllPairsShortestPaths(G)

- 1 Build graph G^* with the extra vertex s^* as above
- 2 **if** $BellmanFord(G^*, s^*) == FALSE$
- 3 Return FALSE # no solution in case of neg. weight cycle
- 4 **else**
- 5 Let $\delta(s^*, v)$ be the results by BellmanFord
- 6 **for all** $v \in V$
- 7 $h(v) = \delta(s^*, v)$
- 8 **for all** $(u, v) \in E$
- 9 $\tilde{w}(u, v) = w(u, v) + h(u) - h(v)$
- 10 **for all** $u \in V$
- 11 run $Dijkstra(\tilde{G}, \tilde{W}, u)$ to obtain $\tilde{d}(u, v)$ for all $v \in V$
- 12 **for all** $v \in V$
- 13 $d(u, v) = \tilde{d}(u, v) - h(u) + h(v)$
- 14 Return $D = (d(u, v))$

Johnson's algorithm (13)

EVERYBODY: COMPUTE THE NEW WEIGHT OF ONE OF THE EDGES IN THE GRAPH:

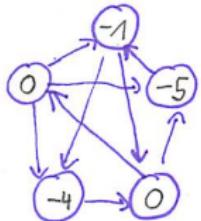
Original graph G :



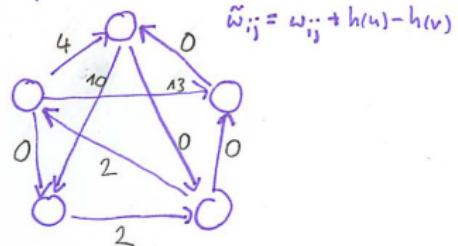
Johnson's algorithm (14)

Solution:

Distances computed by Bellman-Ford:



Graph with new weights:



Johnson's algorithm (15)

Running time of Johnson:

- ▶ One call to BellmanFord $\sim O(V \cdot E)$
- ▶ One loop over vertices $\sim O(V)$
- ▶ One loop over edges $\sim O(E)$
- ▶ **V calls of Dijkstra** $\sim O(VE + V^2 \log V)$ with Fibonacci heaps
- ▶ double loop (vertices and edges, line 12) $\sim O(V \cdot E)$

Together: $O(V \cdot E + V^2 \log V)$

- ▶ If graph is dense ($E = \Theta(V^2)$), then this is pretty bad: $O(V^3)$
- ▶ If graph is very sparse ($E = \Theta(V)$), then this is reasonably good: $O(V^2 \log V)$

Johnson's algorithm (16)

Comment: Johnson's algorithm only makes sense if we want to solve the All-pairs shortest path problem, not for the single pair shortest path problem (why?)

Point to Point Shortest Paths

Point to Point Shortest Paths

Given a graph G and two vertices s and t , we want to compute the shortest path between s and t only.

First idea:

- ▶ Run $Dijkstra(G, s)$ and stop the algorithm when we reached t .
- ▶ This has the same worst case running time as Dijkstra.
- ▶ However, in practice it is often faster because the running time is just in terms of the number of vertices and edges we actually visited.
- ▶ In general, one cannot do better in terms of worst case running time.

Bi-directional Dijkstra

Literature:

I did not find any good textbook reference. A sketch of the algorithm can be found in the following paper:

A. Goldberg, C. Harrelson. Computing the shortest path: A^* search meets graph theory. In: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, pages 156–165, 2005.

Original sources are somewhat hard to track. It has appeared in the literature in the 1970ies.

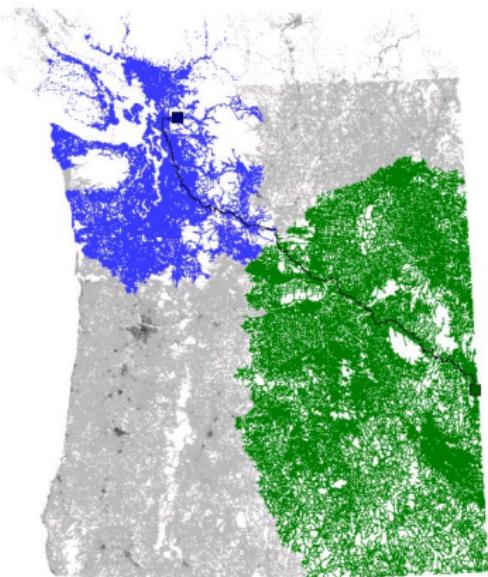
Bi-directional Dijkstra

Idea:

- ▶ Instead of starting Dijkstra at s and waiting until we hit t , we start two copies of the Dijkstra algorithm, one at s (“forward search”) and one at t (“backward search”).
- ▶ We alternate between these two algorithms
- ▶ We stop when the two algorithms “meet”.

Bi-directional Dijkstra (2)

Why should this help? Consider the following example
(taken from slides of Andrew Goldberg):



Left: normal Dijkstra, right: bidirectional Dijkstra

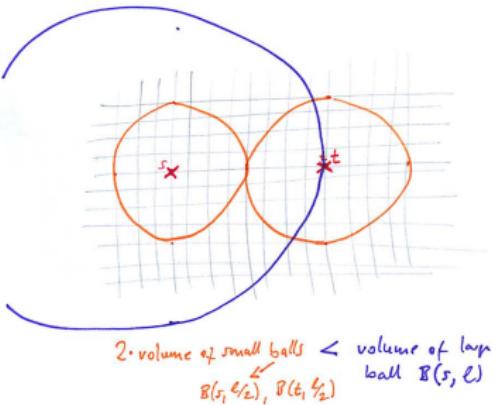
Bi-directional Dijkstra (3)

Observe: Bidirectional \leadsto smaller portion of the graph.

Bi-directional Dijkstra (4)

More formal example:

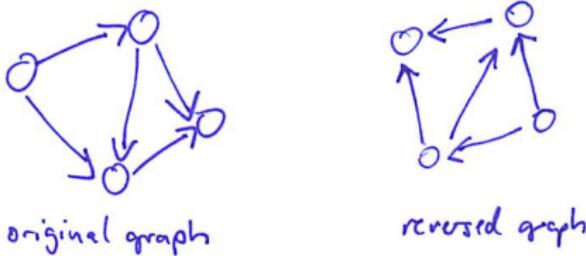
- ▶ Consider s and t as points on a k -dimensional grid, having distance $d(s, t) := \ell$.
- ▶ Standard-Dijkstra visits about $(2k)^\ell$ vertices
- ▶ Bi-directional Dijkstra visits about $2 \cdot k^\ell$ vertices
- ▶ Speedup of a factor 2^{k-1}



Bi-directional Dijkstra (5)

How to perform bidirectional Dijkstra? General idea:

- If the graphs are directed: reverse all edges in the backward search starting from t . $\sim G'$



- Run Dijkstra on G , starting from s , and Dijkstra on G' starting from t , until “they meet”

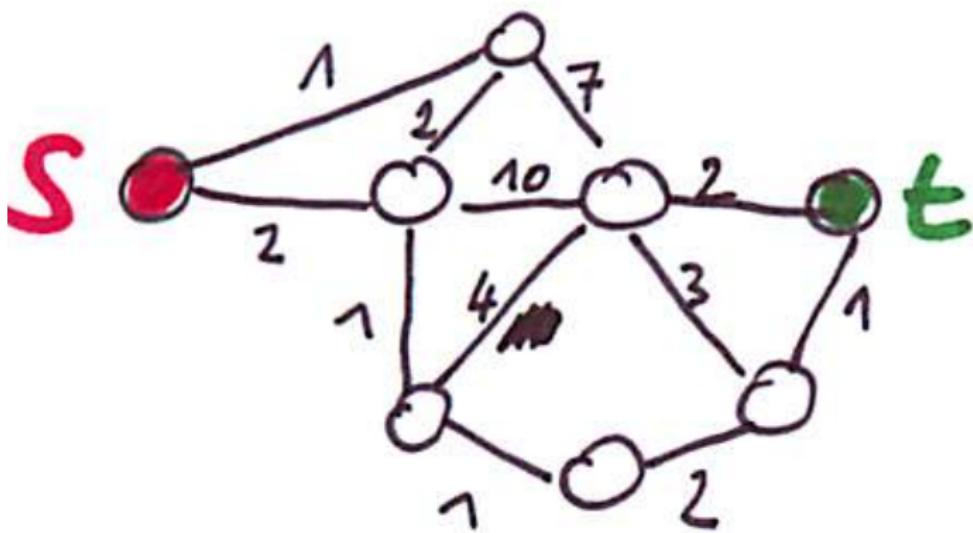
Bi-directional Dijkstra (6)

BidirectionalDijkstra(G, s, t)

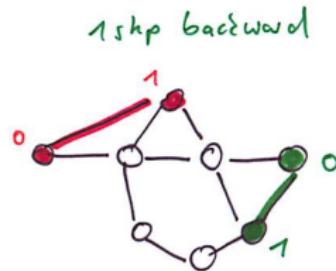
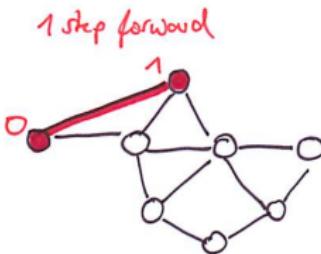
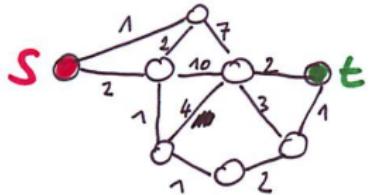
- ▶ $\mu = \infty$ (best path length currently known)
- ▶ Alternately run steps of $Dijkstra(G, s)$ and $Dijkstra(G', t)$
 - ▶ When an edge (v, w) is *scanned* by the forward search and w has already been visited by the backward search:
 - ▶ We have found a path between s and t , namely $s \dots v \ w \dots t$.
 - ▶ The length of this path is $\ell = d(s, v) + w(v, w) + d(w, t)$
 - ▶ If $\mu > \ell$, set $\mu = \ell$.
 - ▶ Analogously for the backward search.
- ▶ We terminate when the search in one direction **selects** a vertex v that has already been selected in the other direction.
- ▶ We return μ as the shortest path length between s and t .

Bi-directional Dijkstra (7)

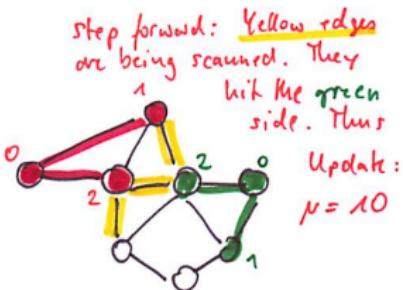
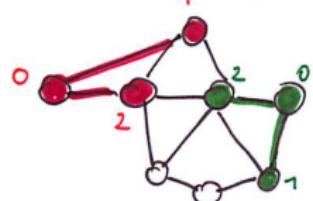
EVERYBODY PLEASE TRY THE FOLLOWING EXAMPLE:



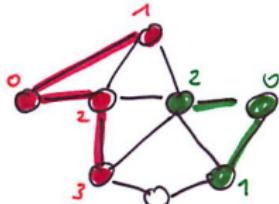
Bi-directional Dijkstra (8)



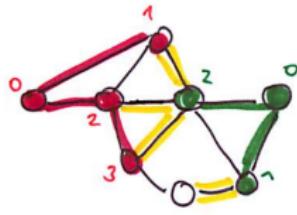
1 step forward, one step backward



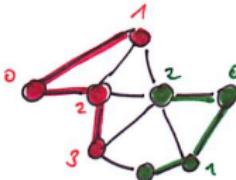
After this step:



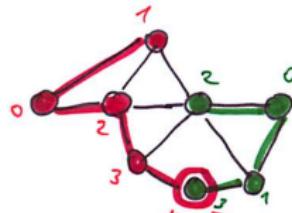
Bi-directional Dijkstra (9)



step backward: Yellow edges
are being examined.
Update $\mu = 9$
(3rd yellow edge from top)



After this step:



Step forward: we select a
vertex that has already
been selected by the
backward search.
Update $\mu = 7$ and stop.

Bi-directional Dijkstra (10)

Theorem 19

If t is reachable from s , then the algorithm

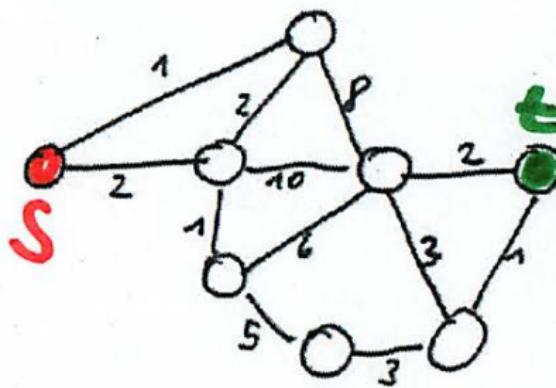
$\text{BidirectionalDijkstra}(G, s, t)$ finds an optimal path, and it is the path stored with μ .

We omit the proof.

Bi-directional Dijkstra (11)

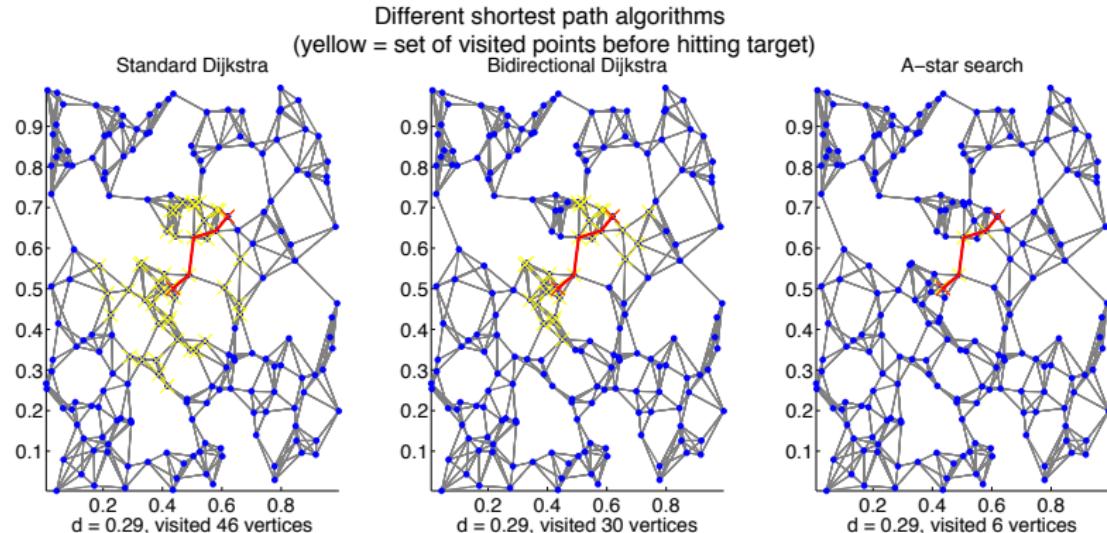
Important remark: It is not always true that if the algorithm stops at v , that then the shortest path between s and t has to go through v !!!

Here is an example graph where this is not the case (...exercise...):



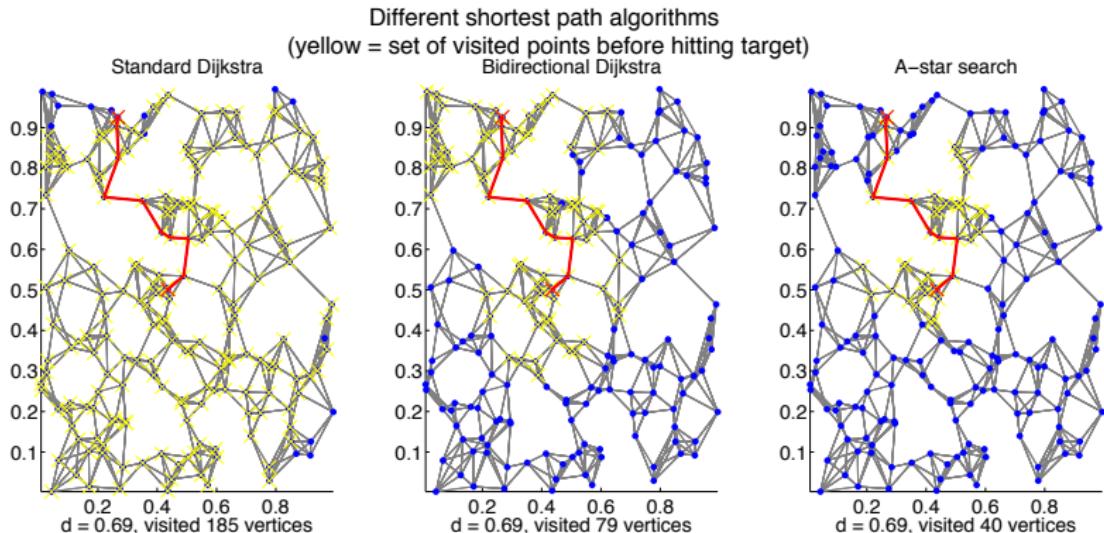
Bi-directional Dijkstra (12)

Example 1: Comparing normal and bidirectional Dijkstra (for now, ignore the third fig.).



Bi-directional Dijkstra (13)

Example 2: Comparing normal and bidirectional Dijkstra (for now, ignore the third fig.).



More elaborate point to point algorithms

.... see A^* heuristics below ...

Minimal spanning trees

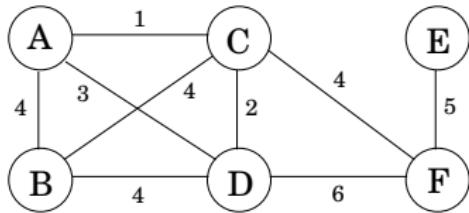
Dasgupta Sec. 5.1; Cormen Sec. 23; Kleinberg Section 4.5;
Mehlhorn/Sanders Section 11;

Problem definition

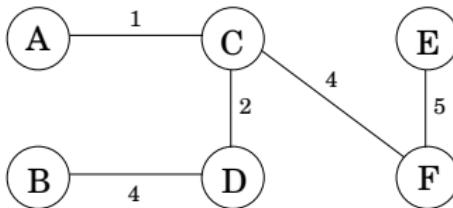
Minimal spanning tree

Intuition:

- ▶ Assume you are given a number of computers, and you want to connect them to a network.
- ▶ Each potential edge has certain cost attached to it.



- ▶ What is the minimal cost to build a connected network?



Minimal spanning tree (2)

Formal problem statement:

Given an undirected graph $G = (V, E)$ with real-valued edge weights $(w_e)_{e \in E}$, find a tree $T = (V', E')$ with $V = V'$, $E' \subset E$ that minimizes

$$\text{weight}(T) := \sum_{e \in E'} w_e.$$

A solution to the problem is called a **minimal spanning tree**, abbreviated **MST**.

Minimal spanning tree (3)

First insights:

- ▶ Minimal spanning trees are not unique, most graphs have many minimal spanning trees.
CAN YOU FIND AN EXAMPLE?
- ▶ If the edge weights are non-negative, then any sub-graph with minimal weight is a tree. CAN YOU SEE WHY?

Minimal spanning tree (4)

HOW WOULD YOU GO ABOUT FINDING A MST?

Generic algorithm

A naive idea

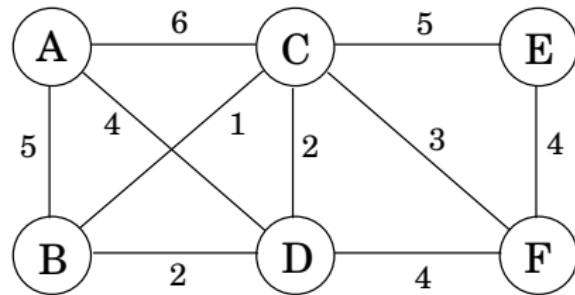
- ▶ Start with an empty tree.
- ▶ Repeatedly add the lightest remaining edge that does not produce a cycle.
- ▶ Stop when the resulting tree connects the whole graph.

Sounds a bit too easy, doesn't it?

A naive idea (2)

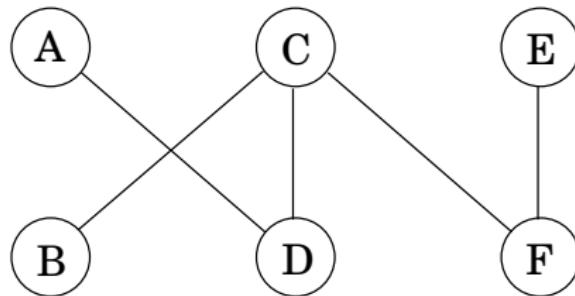
Let's try it.

EVERYBODY, TRY THE FOLLOWING EXAMPLE:



A naive idea (3)

Solution:



Is this really an MST?

A naive idea (4)

Big question: the approach above is a greedy approach.

- ▶ Is it really going to find a minimal spanning tree?
- ▶ Sometimes? Always? Under which conditions????

To answer this question, we take a little detour now.

Safe edges

Given a subset E' of the edges of an MST, a new edge e is called “safe” if there exists a minimal spanning tree with edge set $E' \cup \{e\}$.

Decrypt it:

- ▶ We start with a MST T .
- ▶ We take some of its edges, E' .
- ▶ We now say that a new edge e is safe if we can complete $E' \cup \{e\}$ to an MST T' (which could be identical or different from T).

A generic algorithm

Assume we would know how to construct a safe edge. Then we could run the following algorithm:

GENERIC-MST(G, w)

- 1 $A = \emptyset$
- 2 **while** A does not form a spanning tree
- 3 find an edge (u, v) that is safe for A
- 4 $A = A \cup \{(u, v)\}$
- 5 **return** A

By the definition of “safe”, this algorithm is correct (it maintains the invariant that A is a subset of a MST).

But how to find “safe” edges???

The cut property

Definition: A **cut** $(S, V \setminus S)$ is a partition of the vertex set of a graph in two disjoint subsets.

Theorem 20 (Finding safe edges)

Let $G = (V, E)$ be a connected, undirected graph with real-valued weight function w on the edges. Let A be a subset of E that is included in some minimum spanning tree for G . Let $(S, V \setminus S)$ be a cut such that no edge of A crosses between S and $V \setminus S$. Let e be the lightest edge that crosses between S and $V \setminus S$. Then e is a safe edge.

XXX FIG MST 1 XX

The cut property (2)

Proof.

- ▶ A is subset of some minimal spanning tree T .
- ▶ Either, e is also in T , in which case we are done.
- ▶ So assume that e is not in T . We are now going to construct an alternative minimal spanning tree T' that contains both A and e .
- ▶ Start with all of T and add the edge e to it.
- ▶ This has to produce a cycle in T (because T already contains a path that connects the two end points of e).
- ▶ This cycle must have another edge e' across the cut.
- ▶ Consider T' which is $T \cup \{e\} \setminus \{e'\}$.
- ▶ We now show that T' is an MST as well:

The cut property (3)

- ▶ T' is connected: clear by construction
- ▶ T' is a tree:

Start with T . Remove edge e' . This is going to break T in two pieces, one spanning tree of S , and another spanning tree of $V \setminus S$. Now adding e connects the two pieces, but does not generate a cycle.

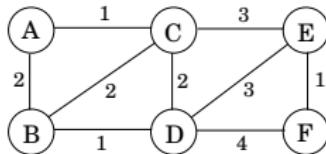
- ▶ T' spans the whole graph: clear
- ▶ T' has minimal costs:

Replacing e' with e can just decrease the costs, by definition of e .

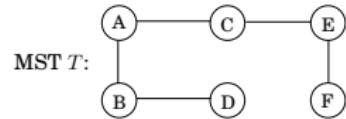
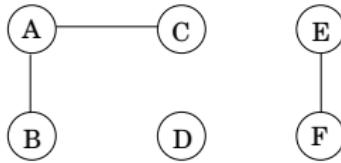


The cut property (4)

Example: Consider the following graph:

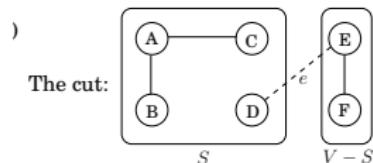


and the following set A (a subset of the MST T):

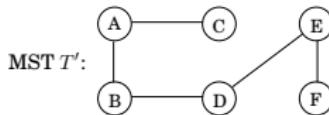


The cut property (5)

Here is a cut (note that A does not cross over the cut):



and here the alternative tree T' constructed in the proof:



Application to the generic algorithm

- ▶ At any point before termination of the algorithm,
 $G_A := (V, A)$ is a forest.
- ▶ We can now repeatedly consider any cut between parts of the forest and add the lightest edge across the cut.
- ▶ This is going to form an MST.

XXX FIG MST 2 XXX

Application to the generic algorithm (2)

EXAMPLE: FIND AN MST IN THE FOLLOWING GRAPH

Application to the generic algorithm (3)

XXX FIG MST 3 XXXX

Kruskal's algorithm, Idea

Kruskal's algorithm, intuition

It is in fact the algorithm we considered in the very beginning:

- ▶ Start with an empty tree.
- ▶ Repeatedly add the lightest remaining edge that does not produce a cycle.
- ▶ Stop when the resulting tree connects the whole graph.

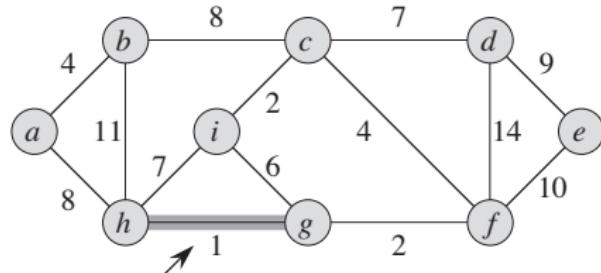
Kruskal's algorithm, naive pseudo-code

KruskalNaive(V,E,W)

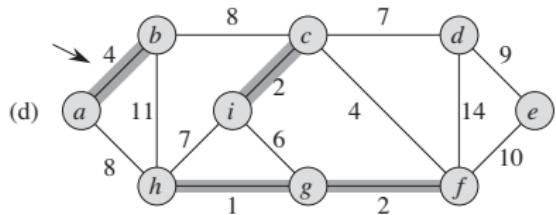
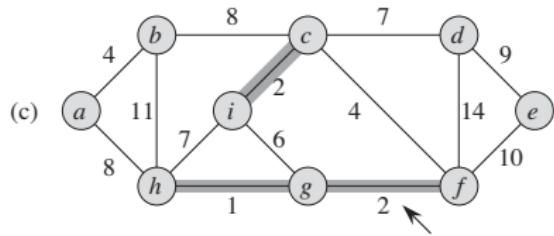
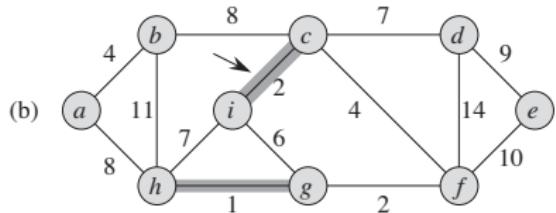
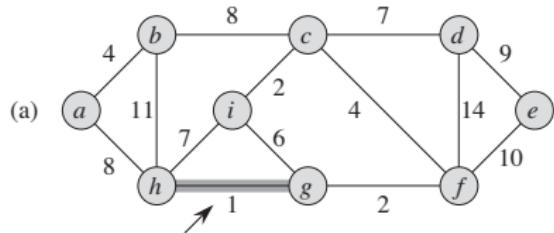
- 1 Sort all edges according to their weight
- 2 $E' = \{\}$
- 3 **for all** $e \in E$, in increasing order of weight
- 4 **if** $E' \cup \{e\}$ does not contain a cycle
- 5 $E' = E' \cup \{e\}$
- 6 **if** E' spans the whole graph
- 7 Return E'

Example

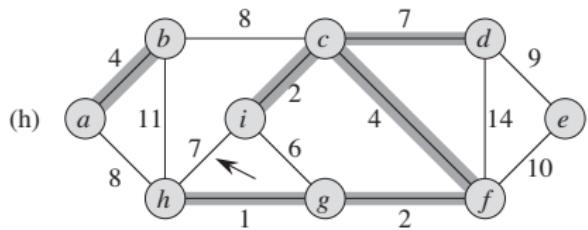
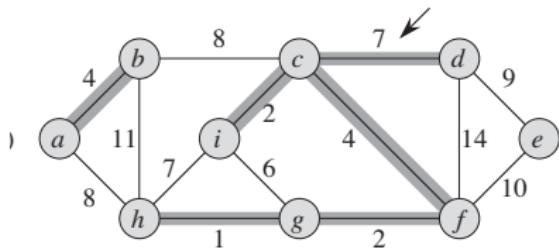
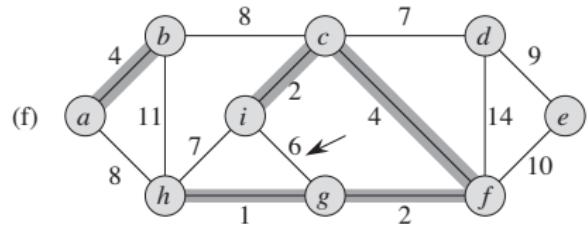
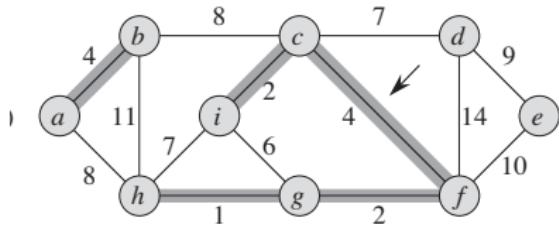
EVERYBODY: RUN KRUSKAL ON THE FOLLOWING EXAMPLE



Example (2)



Example (3)



Implementation, first attempts

Need to execute the following two key operations:

1. Check whether an edge set E' contains a cycle
2. Check whether an edge set E' spans the whole graph.

Running time of the KruskalNavie algorithm:

- ▶ $|E|$ times costs of (1) + $|V|$ times cost of (2)
(WHY EXACTLY?)

ANY IDEAS HOW TO IMPLEMENT (1) AND (2)?

Implementation, first attempts (2)

First idea: adjacency list + BFS.

- ▶ Store the edges in E' as an adjacency list.
- ▶ Implement operation (1) with a BFS.

Running time:

- ▶ In each iteration of the for loop, cost would be $O(E')$ where E' is the current edge set.
- ▶ We have at least V iterations.
- ▶ So all in all the costs of operation (1) would add to $O(1 + 2 + \dots + |V|) = O(|V|^2)$.

Sounds quite expensive already!!!

Implementation, first attempts (3)

- ▶ Operation (2) is easy, we just have to check whether each vertex occurs at least once in the adjacency list. Can do this in $O(n)$ for the whole algorithm.

For example, we could do the following:

- ▶ Array B of size 1 to n , that contains 0 or 1 depending on whether vertex i is contained in the current E' . Counter c that tells us how many vertices are already contained in E' .
- ▶ Whenever we add a new edge to E' , we check in B whether we have seen its vertices already. If not, we update the corresponding entries in B and we increase the counter c accordingly.
- ▶ We stop the algorithm when $c = |V|$.

Implementation, first attempts (4)

Running BFS in each for-loop sounds a bit of an overkill (WHY?).

The idea is now to invent a new data structure that makes it easy to keep track of the connected components and cycles.

Union-find data structure

Kleinberg Sec. 4.6, Cormen Sec. 21, Dasgupta Sec. 5.1.4

Data structure for disjoint sets

Want a data structure that can keep track of “disjoint sets”:

- ▶ Each set gets a “nickname”.
- ▶ For an element u , the operation $\text{Find}(u)$ returns the nickname of the set in which u is contained.
- ▶ We can then easily check whether two elements u, v are in the same set, by asking $\text{Find}(u) \stackrel{?}{=} \text{Find}(v)$.
- ▶ We will implement an efficient way to build the union of two such sets.

Data structure for disjoint sets (2)

Note that if we have such a data structure, we can implement Kruskal:

- ▶ We keep track of the vertices in the subtrees of E'
- ▶ To check whether adding edge e leads to a cycle, we simply ask whether its two end points u, v are in the same subtree or not
- ▶ The algorithm runs until all vertices are connected by one tree.

Abstract Union-Find data structure

Want to maintain the following elementary operations:

- ▶ Makeset(x): create a singleton set $\{x\}$
- ▶ Find(x): return the current “nickname” of the set that contains x
- ▶ Union(x,y): merge the sets that contain x and y

Implementation 1: Naive Arrays

- ▶ Assume you have n elements in total.
- ▶ Take an array *Component* of length n . In entry i , store the nickname of the component that currently contains element i .
XXX UF 1 XXX

Running times:

- ▶ Find: $O(1)$
- ▶ Union: $O(n)$ (WHY?)

Implementation 1: Naive Arrays (2)

Simple improvements:

- ▶ For each of the components, explicitly maintain a list containing the elements.
XXX UF2 XXXX
- ▶ For union, choose the nickname of the united set as the nickname of the larger of the two individual sets.
Then rename the entries in components for all elements in the list of the smaller component.

Running times:

- ▶ Worst case running time of union is still $O(n)$
- ▶ But one can show: Any sequence of k union-operations takes time at most $O(k \log k)$. (see Kleinberg book).

Better implementation: union by rank

- ▶ Each set is stored as a tree (not a binary one!)
- ▶ The name of the set is simply the element in the root
- ▶ Each element stores:
 - ▶ a pointer to its parent (if the vertex is the root, this pointer goes to itself)
 - ▶ for convenience, we also store the “rank” of each element: the height of the subtree hanging below that node

XXX UF 3 XXX

Better implementation: union by rank (2)

MakeSet operation:

```
procedure makeset( $x$ )  
     $\pi(x) = x$   
     $\text{rank}(x) = 0$ 
```

Running time $O(1)$.

Better implementation: union by rank (3)

Find operation:

```
function find(x)  
  while x  $\neq \pi(x)$  : x =  $\pi(x)$   
  return x
```

Running time: height of the current tree.

Better implementation: union by rank (4)

Union operation:

- ▶ make one root point to the root of the other tree.
- ▶ To keep the height of the trees low, make the root of the shorter tree point to the root of the larger tree.

```
procedure union( $x, y$ )
 $r_x = \text{find}(x)$ 
 $r_y = \text{find}(y)$ 
if  $r_x = r_y$ : return
if  $\text{rank}(r_x) > \text{rank}(r_y)$ :
     $\pi(r_y) = r_x$ 
else:
     $\pi(r_x) = r_y$ 
    if  $\text{rank}(r_x) = \text{rank}(r_y)$  :  $\text{rank}(r_y) = \text{rank}(r_y) + 1$ 
```

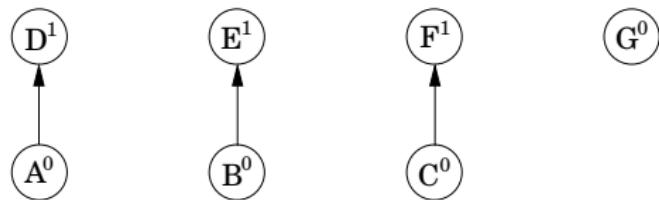
Better implementation: union by rank (5)

Example:

After `makeset(A), makeset(B), ..., makeset(G)`:

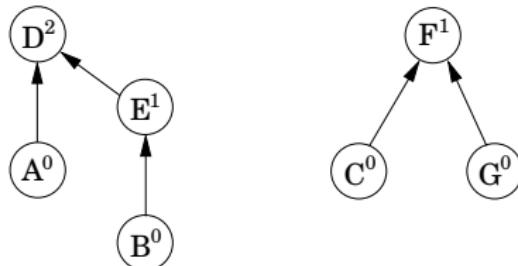


After `union(A, D), union(B, E), union(C, F)`:

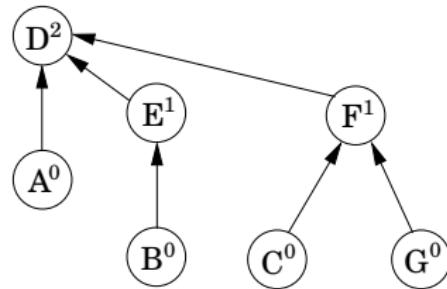


Better implementation: union by rank (6)

After $\text{union}(C, G)$, $\text{union}(E, A)$:



After $\text{union}(B, G)$:



Union-by-rank: analysis

Proposition 21 (Running time of union and find)

In the union-by-rank implementation, both Union and Find-operation take worst case running time $O(\log n)$ where n is the total number of elements.

The proof takes a couple of steps:

Property 1: For any x , we have $\text{rank}(x) < \text{rank}(\pi(x))$.

Reason: Obvious.

Union-by-rank: analysis (2)

Property 2: Any node of rank k has at least 2^k descendants.

Reason: Induction, exercise (TRY IT!).

Property 3: If there are n elements overall, there can be at most $n/2^k$ nodes of rank k .

Reason: direct consequence of Property 2.

Union-by-rank: analysis (3)

Property 4: The maximum over all ranks is at most $\log n$, that is all trees have height at most $\log n$.

Reason: Direct consequence of Property 3.



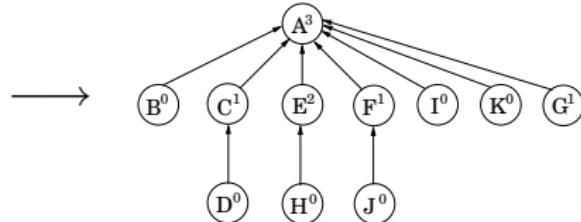
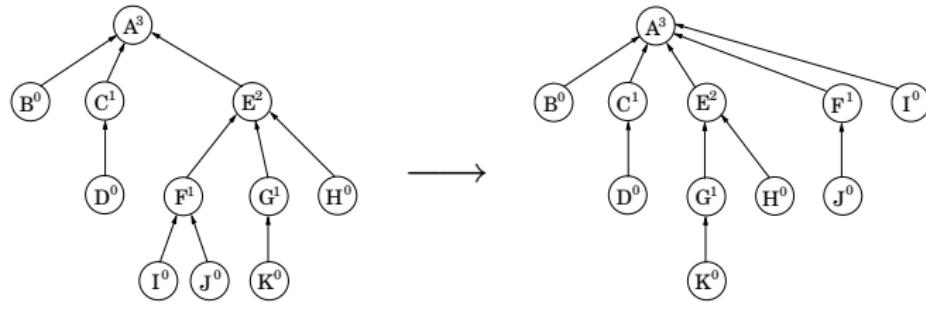
Further improving union-rank: path compression

Want to keep the height of the trees even shorter. Here is the idea:

- ▶ During each find-operation, we have to follow the pointers from x up to the root.
- ▶ We now make all the vertices along this path point to the root directly.
- ▶ This makes this particular run of *find* a little more expensive, but when we later call *find* on any of the intermediate vertices, it is much cheaper.

Further improving union-rank: path compression (2)

- ▶ Example: Find(I), followed by Find(K)



Further improving union-rank: path compression (3)

- ▶ One can prove that if we look at any sequence of m find-operations, then the “amortized cost” (\approx average cost) for each of these operations is just a bit larger than $O(1)$.). See Dasgupta Sec. 5.1.2 or Cormen Sec. 21.4. for details.

Kruskal's algorithm, revisited

Kruskal's algorithm with union-find data structure

MST-KRUSKAL(G, w)

- 1 $A = \emptyset$
- 2 **for** each vertex $v \in G.V$
 - 3 MAKE-SET(v)
 - 4 sort the edges of $G.E$ into nondecreasing order by weight w
 - 5 **for** each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
 - 6 **if** FIND-SET(u) \neq FIND-SET(v)
 - 7 $A = A \cup \{(u, v)\}$
 - 8 UNION(u, v)
 - 9 **return** A

Kruskal's algorithm with union-find data structure (2)

Running time with the union-by-rank implementation:

- ▶ $|V|$ calls to $\text{MakeSet}(v)$, gives $O(V)$
- ▶ Sorting in line 4, takes $O(E \log E) = O(E \log V)$
- ▶ in the loop, $2E$ find-set and $V - 1$ union-operations, where each of these operations takes $\log V$.

In total: $O(V + E \log V + (E + V) \log V) = O(E \log V)$.

Prim's algorithm

General idea

Intermediate set of edges always forms a tree:

- ▶ Denote by S the set of vertices that are connected by the current tree. We initialize S to contain some starting vertex s .
- ▶ As long as $S \neq V$, consider all edges that connect a vertex in S to a vertex in $V \setminus S$. Among those edges, take the cheapest one and add the corresponding end vertex to S .

General idea (2)

DOES IT RING A BELL?

General idea (3)

Looks pretty much like Dijkstra's algorithm for the shortest path...

As for Dijkstra, the best way to implement this algorithm is to use a priority queue. Here is the pseudocode:

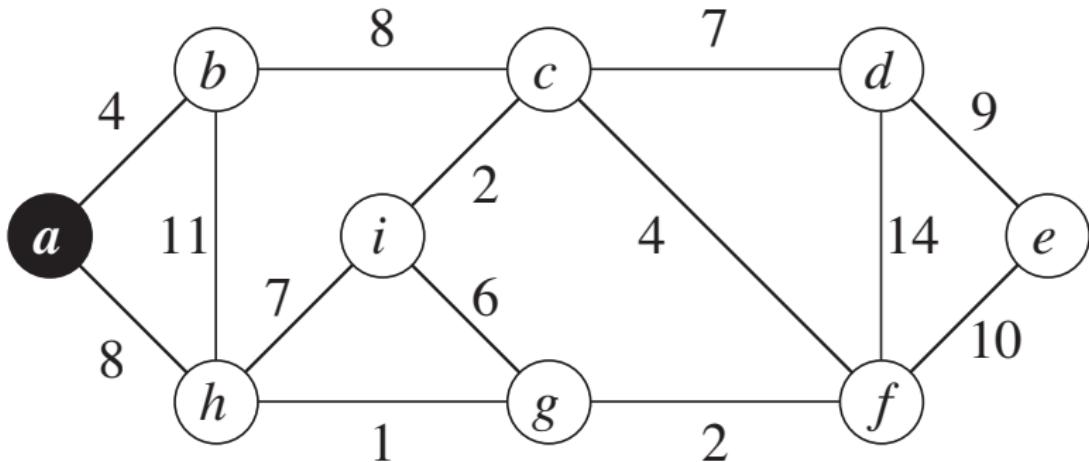
General idea (4)

MST-PRIM(G, w, r)

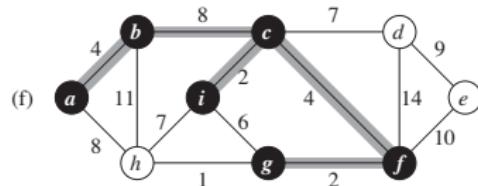
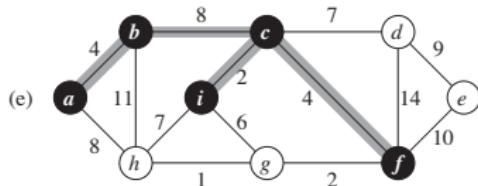
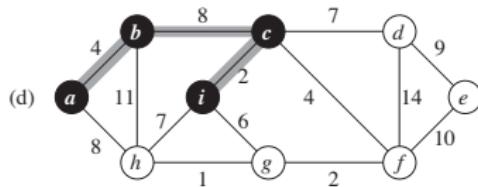
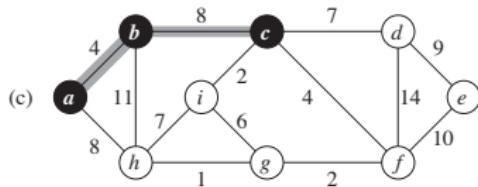
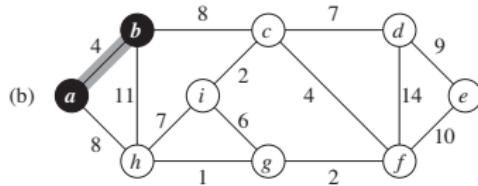
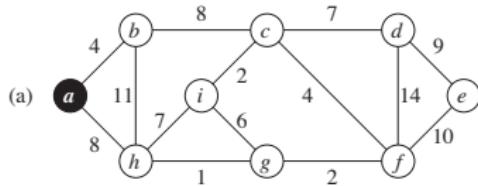
- 1 **for** each $u \in G.V$
- 2 $u.key = \infty$
- 3 $u.\pi = \text{NIL}$
- 4 $r.key = 0$
- 5 $Q = G.V$
- 6 **while** $Q \neq \emptyset$
- 7 $u = \text{EXTRACT-MIN}(Q)$
- 8 **for** each $v \in G.Adj[u]$
- 9 **if** $v \in Q$ and $w(u, v) < v.key$
- 10 $v.\pi = u$
- 11 $v.key = w(u, v)$

Prim's algorithm: example

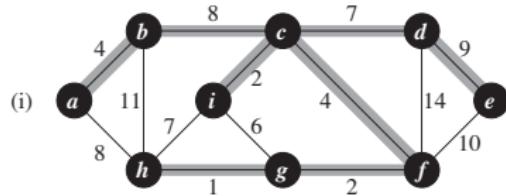
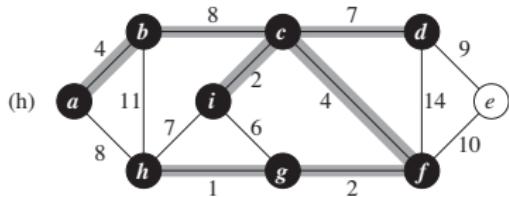
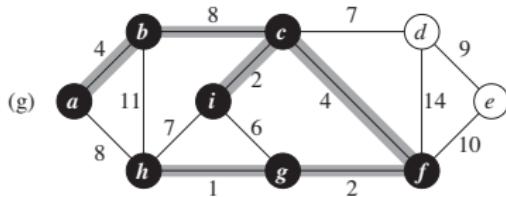
EVERYBODY: TRY THE FOLLOWING EXAMPLE (start in vertex a):



Prim's algorithm: example (2)



Prim's algorithm: example (3)



Prim's algorithm: analysis

Correctness: follows from the safe edge theorem.

WHY EXACTLY?

Prim's algorithm: analysis (2)

Running time: Exercise!

Result is $O(E \log V)$, same as for Kruskal's algorithm.

History / Original sources

Kruskal's algorithm:

- ▶ J. Kruskal: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society, 1956

Prim's algorithm:

- ▶ R. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, 1957.
- ▶ Apparently had already been invented by V. Jarnik in 1930.

Advanced methods were published in the 1980ies, e.g. by Tarjan and coworkers, and even more improvements in 2000 by Chazelle.