Introduction to Algorithms

Chapter 1: Basics on algorithms

Manuel

Fall 2017

Outline

Algorithm

Algorithm: Recipe telling the computer how to solve a problem.

Algorithm

Algorithm: Recipe telling the computer how to solve a problem.

Example.
I am the "computer", detail an algorithm such that I can prepare a jam sandwich.

Actions: cut, listen, spread, sleep, read, take, eat, dip
Things: knife, guitar, bread, honey, jam jar, sword

Algorithm

Algorithm: Recipe telling the computer how to solve a problem.

Example.
I am the "computer", detail an algorithm such that I can prepare a jam sandwich.

Actions: cut, listen, spread, sleep, read, take, eat, dip
Things: knife, guitar, bread, honey, jam jar, sword

Algorithm. (*Sandwich making*)

Input : 1 bread, 1 jamjar, 1 knife
Output: 1 jam sandwich

1 take the knife and cut 2 slices of bread;
2 dip the knife into the jamjar;
3 spread the jam on the bread, using the knife;
4 assemble the 2 slices together, jam on the inside;

A more formal view

An algorithm systematically solves a *well-defined* problem:

- The *Input* is clearly expressed
- The *Output* solves the problem
- The *Algorithm* provides a precise step-by-step procedure starting from the Input and leading to the Output

A more formal view

An algorithm systematically solves a *well-defined* problem:

- The *Input* is clearly expressed
- The *Output* solves the problem
- The *Algorithm* provides a precise step-by-step procedure starting from the Input and leading to the Output

Algorithms can be described using one of the three following ways:

- English
- Pseudocode
- Programming language

A first example

Algorithm. (*Insertion Sort*)

Input : $a_1, \ldots, a_n$, n unsorted elements
Output: the $a_i, 1 \leq i \leq n$, in increasing order

for $j \leftarrow 2$ **to** n **do**
    $i \leftarrow 1$;
    while $a_j > a_i$ **do** $i \leftarrow i + 1$;
    $m \leftarrow a_j$;
    for $k \leftarrow 0$ **to** $j - i - 1$ **do** $a_{j-k} \leftarrow a_{j-k-1}$;
    $a_i \leftarrow m$
end for
return $(a_1, \ldots, a_n)$

A first problem

Setup: a robot arm solders chips on a board in n contact points

Goal: minimize the time to attach a chip to the board

Assumptions:

- The arm moves at constant speed
- Once a chip has been attached another one is soldered

A first problem

Setup: a robot arm solders chips on a board in n contact points

Goal: minimize the time to attach a chip to the board

Assumptions:

- The arm moves at constant speed
- Once a chip has been attached another one is soldered

Defining the Input and Output:

- Input: a set S of n points in the plane
- Output: the shortest path visiting all the points in S

A first solution

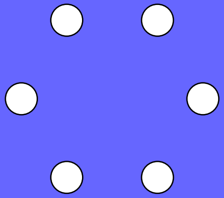
Algorithm. (*Nearest neighbor*)

Input : a set $S = \{s_0, \cdots, s_{n-1}\}$ of n points in the plane
Output: the shortest cycle visiting all the points in S

$p_0 \leftarrow s_0$;
for $i \leftarrow 1$ **to** $n-1$ **do**
    $p_i \leftarrow$ closest unvisited neighbor to p_{i-1};
    Visit p_i;
end for
return $\langle p_0, \ldots, p_{n-1} \rangle$

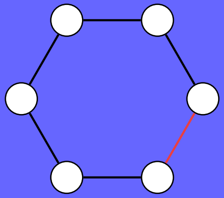
A first failure

How does the nearest neighbor algorithm (1.7) perform in the following three cases?



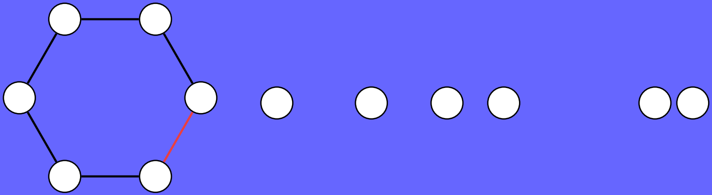
A first failure

How does the nearest neighbor algorithm (1.7) perform in the following three cases?



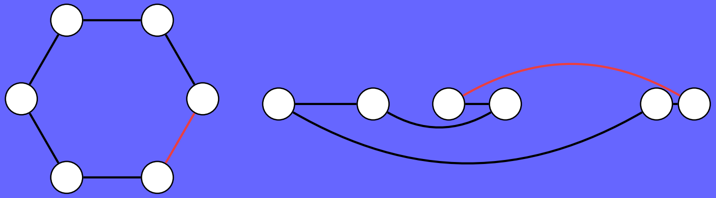
A first failure

How does the nearest neighbor algorithm (1.7) perform in the following three cases?



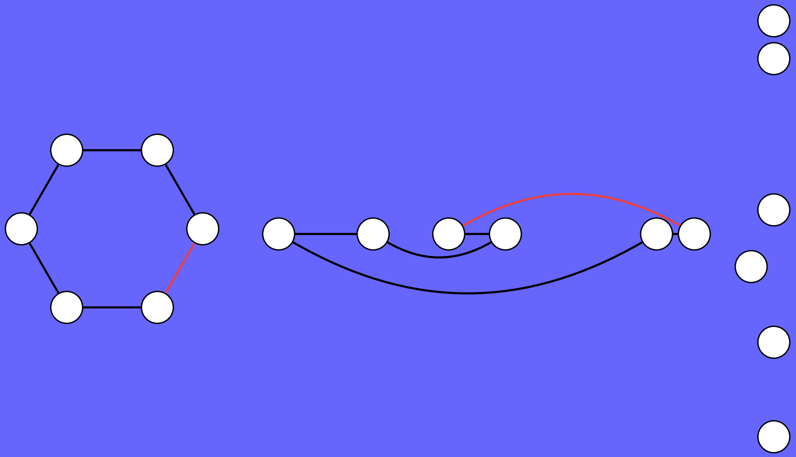
A first failure

How does the nearest neighbor algorithm (1.7) perform in the following three cases?



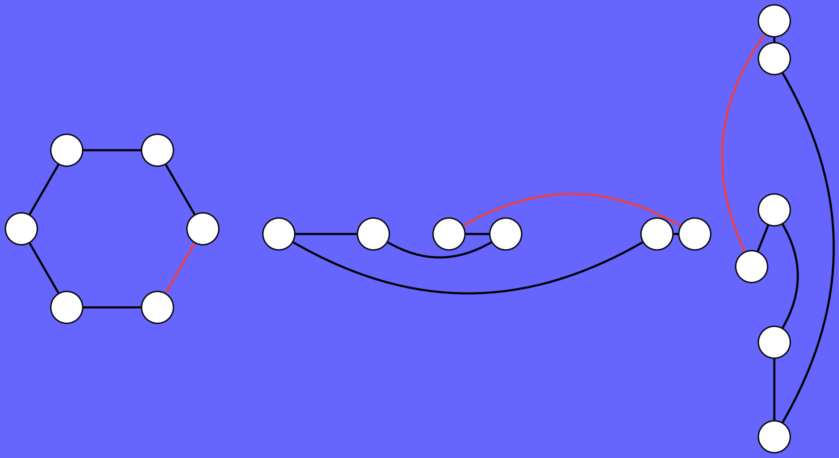
A first failure

How does the nearest neighbor algorithm (1.7) perform in the following three cases?



A first failure

How does the nearest neighbor algorithm (1.7) perform in the following three cases?



A second solution

Algorithm. (*Closest pair*)

Input : a set S of n points in the plane
Output: the shortest cycle visiting all the points in S

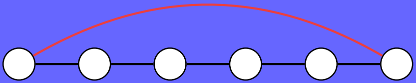
for $i \leftarrow 1$ **to** $n - 1$ **do**
  $d \leftarrow \infty$;
  foreach *pair of end points* $\langle s, t \rangle$ *from distinct vertex chains* **do**
    if $dist(s, t) \leq d$ **then**
      $s_m \leftarrow s$; $t_m \leftarrow t$; $d \leftarrow dist(s, t)$;
    end if
  end foreach
  Connect s_m and t_m by an edge;
end for
return *all the points starting from one of the two end points*

A second failure

Applying the closest pair algorithm (1.9) on the following vertices arrangement yields the two graphs:

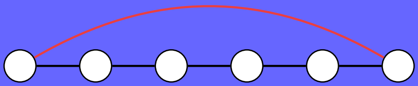
A second failure

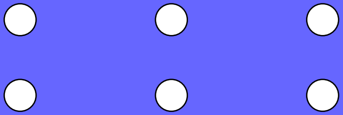
Applying the closest pair algorithm (1.9) on the following vertices arrangement yields the two graphs:



A second failure

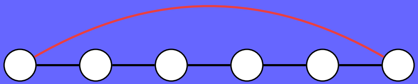
Applying the closest pair algorithm (1.9) on the following vertices arrangement yields the two graphs:

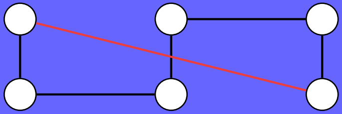




A second failure

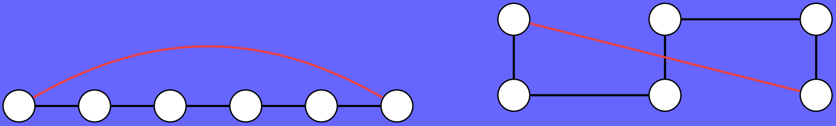
Applying the closest pair algorithm (1.9) on the following vertices arrangement yields the two graphs:





A second failure

Applying the closest pair algorithm (1.9) on the following vertices arrangement yields the two graphs:



Possible strategy to ensure the most optimal path:

- Enumerate all the possible paths
- Select the one that minimizes the total length

Drawback: for only 20 vertices $20! = 2432902008176640000$ paths have to be explored...

The first few lessons

A difference:

- Algorithm: always output a correct result
- Heuristic: idea serving as a guide to solve a problem with no guarantee of always providing a good solution

Correctness and efficiency:

- An algorithm working on a set of input does not imply it will work on all instances
- Efficient algorithm totally solving a problem might not exist

Solving problems using algorithms

Common traps when defining the Input and Output:

- Are they precise enough?
- Can all the Input be easily and efficiently generated?
- Could there be any confusion on the expected Output?

Example.
For an Output, what does it mean to “find the best route”?

Solving problems using algorithms

Common traps when defining the Input and Output:

- Are they precise enough?
- Can all the Input be easily and efficiently generated?
- Could there be any confusion on the expected Output?

Example.

For an Output, what does it mean to “find the best route”?

The shortest in distance, the fastest in time, or the one minimizing the number of turns?

Conclusion: where to start (Input) and where to go (Ouput) must be expressed in simple, precise, and clear terms.

Incorrectness

Finding good counter-examples:

- Seek simplicity: make it clear why the algorithm fails
- Think small: algorithms failing for large Input often fail for smaller one
- Test the extremes: study special cases, e.g. inputs equal, tiny, huge...
- Think exhaustively: test whether all the possible cases are covered by the algorithm
- Track weaknesses: check if the underlying idea behind the algorithm has any "unexpected" impact on the output

Outline

Continuous vs. linked

Data structures can be split into two main categories:

- **Continuous:** a single piece of memory
  e.g. array, matrices, hash tables...
- **Linked data structures:** distinct chunks of memory connected together
  e.g. linked list, trees, graph adjacency lists...

Each type has some relative advantages. Choosing the proper data type is of a major importance when designing algorithms.

Arrays

Each element can be efficiently located using its index:

- **Constant access time:** each index maps to a memory address
- **Space efficiency:** no space wasted with links or information on the data
- **Memory locality:** data is contiguous so *cache* can be used to speed up successive data accesses

Downside: the size cannot be easily adjusted during the program's execution

Linked structures

A linked structure is composed of nodes. Each one contains:

- One or more fields on data
- A pointer to at least another node

The most common operations are:

- Search: find an item in the list
- Insert: add an item to the list
- Delete: remove an item from the list

Search can be implemented either iteratively or recursively

Comparison

Linked structure

- Overflow only occurs when memory is full
- Insertion/deletion are simple and fast
- Moving pointers is faster than moving the actual data

Array

- No extra space wasted for the pointer field
- Efficient random access is possible
- Better memory locality and cache performance

Containers

Common data structures allowing the storage and retrieval of data independently of the content:

- **Stack:**
  - LIFO order
  - Simple to implement and very efficient
- **Queue:**
  - FIFO order
  - Minimize the maximum waiting time
  - Trickier to implement than stacks

Both can be implemented using either linked lists or arrays, depending if the size of the container is known in advance

Dictionaries

Data type allowing access by content. Primary operations:

- **Search:** search a value in a given dictionary
- **Insert:** add an element to the dictionary
- **Delete:** remove an element from the dictionary

Most common operations:

- **Max/Min:** retrieve the largest/smallest element from the dictionary
- **Predecessor/Successor:** retrieve the element just before/after a given element; before/after are defined with respect to a sorted order

Dictionary using arrays

Let n be the number of elements in the array

Operation	Unsorted array	Sorted array
`search(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
`insert(D,k)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`delete(D,k)`	$\mathcal{O}(1)^*$	$\mathcal{O}(n)$
`predecessor(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`successor(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`minimum(D)`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`maximum(D)`	$\mathcal{O}(n)$	$\mathcal{O}(1)$

* Assuming a pointer to the key k is given how to get $\mathcal{O}(1)$?

Dictionary using linked structures

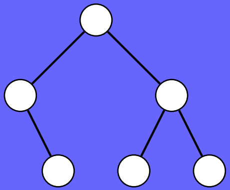
Let n be the number of elements in the list

Operation	Unsorted		Sorted	
	Single	Double	Single	Double
`search(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`insert(D,k)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`delete(D,k)`	$\mathcal{O}(n)^*$	$\mathcal{O}(1)$	$\mathcal{O}(n)^*$	$\mathcal{O}(1)$
`predecessor(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)^*$	$\mathcal{O}(1)$
`successor(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`minimum(D)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`maximum(D)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)^\dagger$	$\mathcal{O}(1)$

* Why are singly linked lists slower?

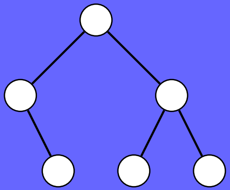
† How to achieve $\mathcal{O}(1)$ for singly sorted lists?

Binary search trees



- Constructed from doubly linked list
- First object is the root of the tree
- Second object is a left child if it precedes the root and a right child if it succeeds it
- Third and further object are sorted along the tree following a similar pattern

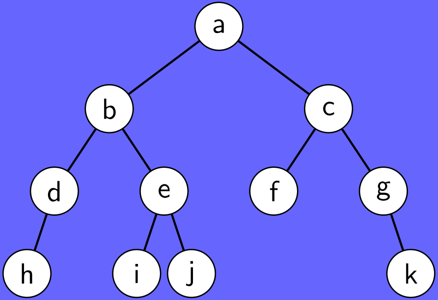
Binary search trees



- Constructed from doubly linked list
- First object is the root of the tree
- Second object is a left child if it precedes the root and a right child if it succeeds it
- Third and further object are sorted along the tree following a similar pattern

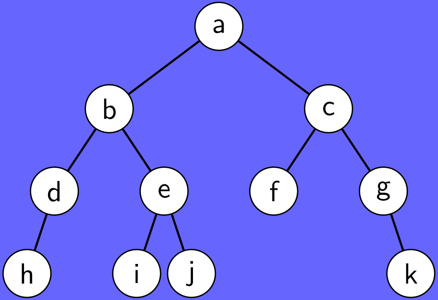
- The three primary dictionary operations take $\mathcal{O}(h)$, with h the height of the tree
- Binary search trees balance the search time and flexible update
- How to handle deletion?

Binary search tree traversal



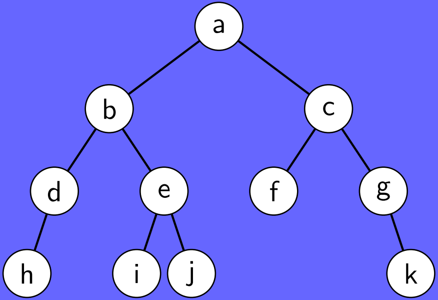
- Preorder traversal:

Binary search tree traversal



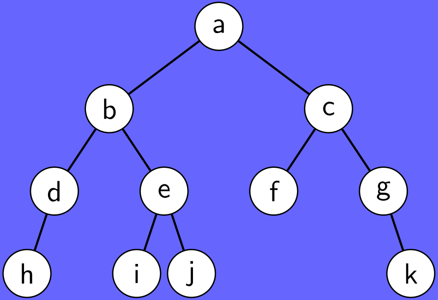
- Preorder traversal:
  a, b, d, h, e, i, j, c, f, g, k
- Inorder traversal:

Binary search tree traversal



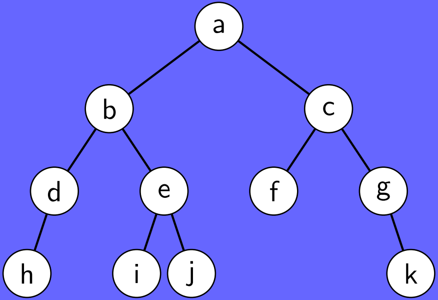
- Preorder traversal:
  a, b, d, h, e, i, j, c, f, g, k
- Inorder traversal:
  h, d, b, i, e, j, a, f, c, g, k
- Postorder traversal:

Binary search tree traversal



- Preorder traversal:
  a, b, d, h, e, i, j, c, f, g, k
- Inorder traversal:
  h, d, b, i, e, j, a, f, c, g, k
- Postorder traversal:
  h, d, i, j, e, b, f, k, g, c, a

Binary search tree traversal



- Preorder traversal:
  a, b, d, h, e, i, j, c, f, g, k
- Inorder traversal:
  h, d, b, i, e, j, a, f, c, g, k
- Postorder traversal:
  h, d, i, j, e, b, f, k, g, c, a

How to implement inorder tree traversal?

Priority queue

Primary operations for priority queues:

- **Insert:** add an element to the queue
- **Find minimum/maximum:** return the last/first element in the queue
- **Delete minimum/maximum:** remove the last/first element in the queue

Operation	Array		Balanced tree
	Unsorted	Sorted	
`insert(Q,x)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
`find_min(Q)`	$\mathcal{O}(1)^*$	$\mathcal{O}(1)$	$\mathcal{O}(1)^*$
`delete_min(Q)`	$\mathcal{O}(n)$	$\mathcal{O}(1)^\dagger$	$\mathcal{O}(\log n)$

* How to reach $\mathcal{O}(1)$ for an unsorted array and a balanced tree?

† How to reach $\mathcal{O}(1)$ when deleting the min in a sorted array?

Hash tables

Practical way to maintain a dictionary where:

- The data is stored in an array
- Each key is hashed and stored at index “the hash of the key”
- Keys with a similar hash are store in a linked list

Good hash function: all indices occur with equiprobability

Hash tables

Practical way to maintain a dictionary where:

- The data is stored in an array
- Each key is hashed and stored at index "the hash of the key"
- Keys with a similar hash are store in a linked list

Good hash function: all indices occur with equiprobability

Example.
A common choice is $H(k) = k \bmod m$, with m a prime not too close from a power of 2.

For $n = 2000$, 701 would be a good choice if one desires to have about three keys stored at each index.

Other common data structures

- **Strings:** array of characters; use suffix trees/arrays for pattern matching
- **Geometric element:** define regions as polygons using segments and points in an array or a tree
- **Graphs:** consider the adjacency matrix or an adjacency list; graph algorithms vary depending on the structure
- **Sets:** bit vector where the element in the set is the index and the value store is 1 or 0 depending whether the element is in the set; dictionaries can be used for fast membership queries

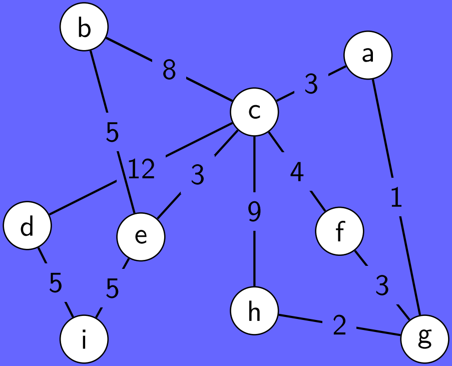
Summary

A few points to remember when selecting a data structure:

- Data can be represented in many ways
- No data structure is fast in all aspects
- Choosing the wrong data structure can be disastrous in terms of performance
- Several choices are often possible
- Identifying the best data structure is often not critical
- Always aim for clear, simple, and efficient data structures

Outline

A first graph problem



Simple problem:

- We have *n* computers connected by wires
- Using different wires implies different costs
- We want:
  - All the computers to be connected to the network
  - Minimize the cost

Minimum spanning tree

Problem (Minimum Spanning Tree (MST))

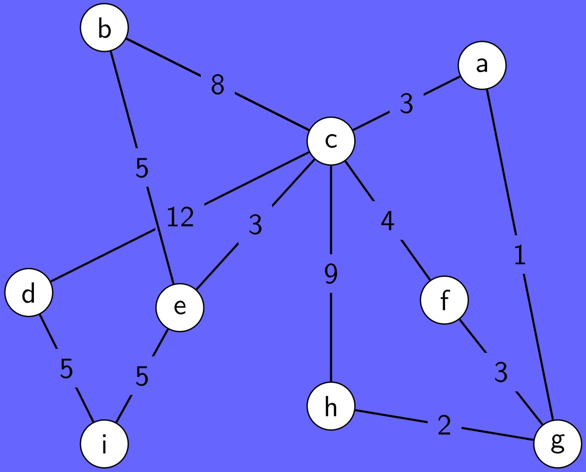
Given a weighted graph G, find a subgraph T such that:

1. All the vertices on G are connected on T,
2. The total weight, defined as the sum of the weight of all the edges in T, is minimized.

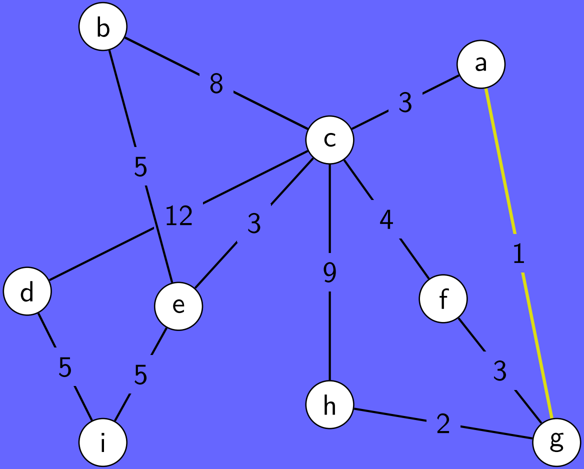
The graph T is a *minimum spanning tree* for G.

Note that T is a tree: if it contained a cycle, at least one edge could be removed, allowing a lower weight while preserving the connected property of T ↯.

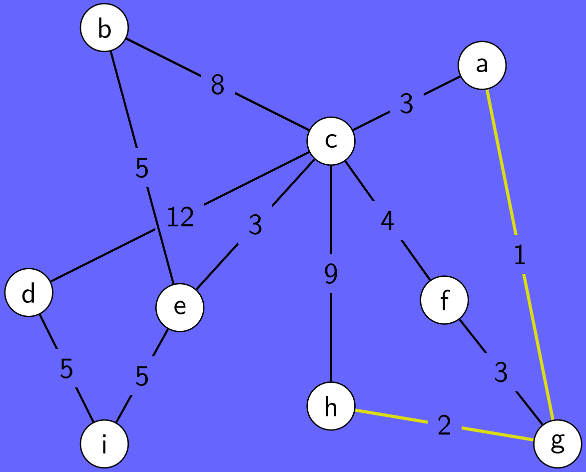
A greedy approach



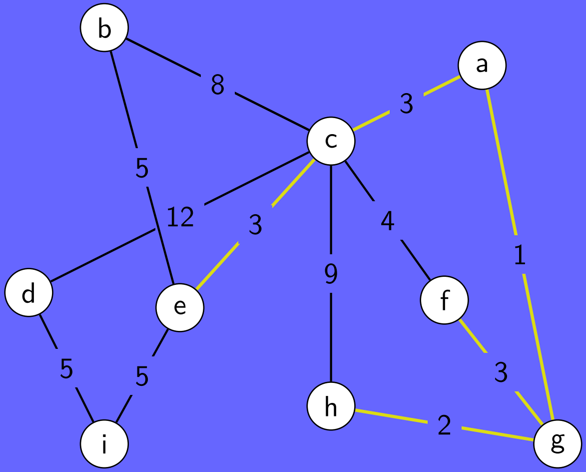
A greedy approach



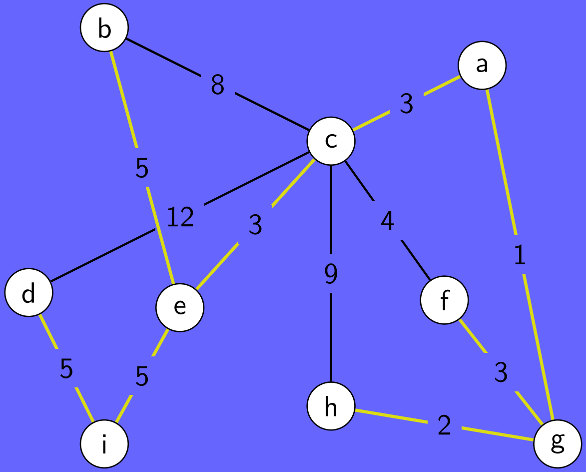
A greedy approach



A greedy approach



A greedy approach



Kruskal's algorithm

Algorithm. (*Kruskal*)

Input : A graph $G = \langle V, E \rangle$
Output: A minimum spanning tree T for G

```
Sort the edges G.E by weight;
T ← ∅;
for edges (u, v) in G.E, in non-decreasing order do
    if adding (u, v) does not create a cycle then
        add edge (u, v) to T
    end if
end for
return T
```

Kruskal's algorithm

Algorithm. (*Kruskal*)

Input : A graph $G = \langle V, E \rangle$
Output: A minimum spanning tree T for G

Sort the edges $G.E$ by weight;
$T \leftarrow \emptyset$;
for *edges* (u, v) *in* $G.E$*, in non-decreasing order* **do**
  if *adding* (u, v) *does not create a cycle* **then**
    add edge (u, v) to T
  end if
end for
return T

What needs to be specified?

Correctness of Kruskal's algorithm

Theorem

Assuming the previous notations, Kruskal's algorithm produces a minimum spanning tree for G.

Proof.

Let $G = \langle V, E \rangle$ be a graph and let v and w be two vertices connected by an edge. If S is the set of all the vertices with a path to v before e is added, then $w \notin S$, otherwise this would define a cycle. Moreover if there was an edge with smaller weight than e, connecting S and $V - S$, then it would have already been added. Therefore e is the cheapest edge connecting $V - S$ to S, and as such belongs to a minimum spanning tree of G.
Clearly by design the algorithm will not generate any cycle. Moreover as G is connected and all the edges are explored $V - S$ and S will be linked at some stage. Hence T is connected. $\square$

Back to the algorithm

Issue: how to represent the data such that whether or not adding an edge creates a cycle can be efficiently tested?

For each edge joining two vertices v and w:

- Identify all the connected components of v and w
- If the edge is to be included, merge the two components

Back to the algorithm

Issue: how to represent the data such that whether or not adding an edge creates a cycle can be efficiently tested?

For each edge joining two vertices v and w:

- Identify all the connected components of v and w
- If the edge is to be included, merge the two components

Extra notes:

- No edge needs to be removed
- No component needs to be split
- Everything must be done efficiently

Toward a new data structure

Representing data using:

- **An array:** testing can be done in constant time; merging requires linear time
- **A graph:** merging is only adding an edge; testing requires a full graph traversal

Toward a new data structure

Representing data using:

- **An array:** testing can be done in constant time; merging requires linear time
- **A graph:** merging is only adding an edge; testing requires a full graph traversal

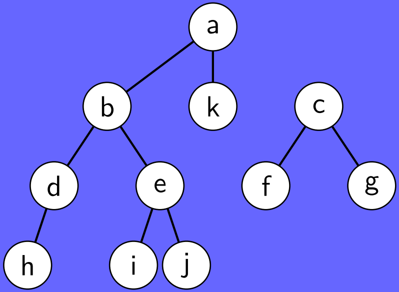
Implement a new data structure containing:

- A pointer to the parent
- The *rank*, or depth, of the sub-tree

The union-find data structure

The two operations are:

- `Find(v)`: find the root of the tree for vertex v
- `Union(v,w)`: link the root of the tree containing v to the root of the tree containing w (or the other way around)

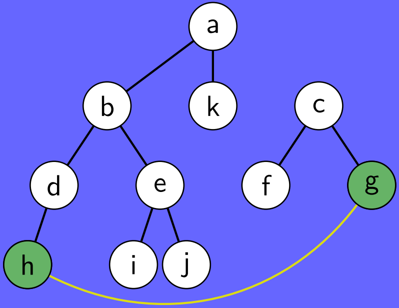


- We have two sub-trees

The union-find data structure

The two operations are:

- `Find(v)`: find the root of the tree for vertex v
- `Union(v,w)`: link the root of the tree containing v to the root of the tree containing w (or the other way around)

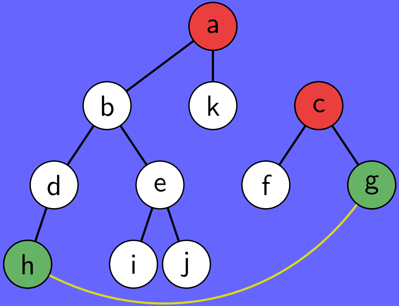


- We have two sub-trees
- On the graph an edge joins the vertices h and g

The union-find data structure

The two operations are:

- `Find(v)`: find the root of the tree for vertex v
- `Union(v,w)`: link the root of the tree containing v to the root of the tree containing w (or the other way around)

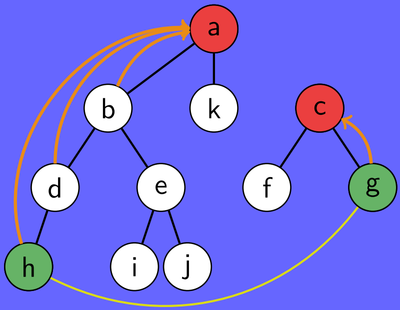


- We have two sub-trees
- On the graph an edge joins the vertices h and g
- `Find` on h and g returns a and c respectively

The union-find data structure

The two operations are:

- `Find(v)`: find the root of the tree for vertex v
- `Union(v,w)`: link the root of the tree containing v to the root of the tree containing w (or the other way around)

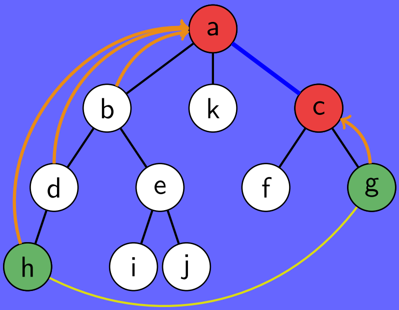


- We have two sub-trees
- On the graph an edge joins the vertices h and g
- `Find` on h and g returns a and c respectively
- Update parents for h, d, b and g

The union-find data structure

The two operations are:

- `Find(v)`: find the root of the tree for vertex v
- `Union(v,w)`: link the root of the tree containing v to the root of the tree containing w (or the other way around)



- We have two sub-trees
- On the graph an edge joins the vertices h and g
- `Find` on h and g returns a and c respectively
- Update parents for h, d, b and g
- `Union` connects c to a

Handling a union-find structure

Algorithm.

```
Function GenSet(x):
    x.parent ← x; x.rank ←0;
end
Function Find(x):
    if x.parent ≠ x then  x.parent ← Find(x.parent) ;
    return x.parent
end
Function Union(x,y):
    X ← Find(x); Y ← Find(y);
    if X.rank > Y.rank then Y.parent ← X;
    else X.parent ← Y;
    if X.rank = Y.rank then Y.rank++;
end
```

Revisiting Kruskal's algorithm

Algorithm. (*Kruskal with find-union*)

Input : A graph $G = \langle V, E \rangle$
Output: A minimum spanning tree T

```
Sort the edges G.E by weight;
T ← ∅;
for edges (u, v) in G.E, in non-decreasing order do
    if Find(u) ≠ Find(v) then
        add edge (u, v) to T;
        Union(u,v)
    end if
end for
return T
```

Counting inversions

Problem (Counting inversions)

Given a list of n elements $a_0, \cdots, a_{n-1}$, determine how many pairs $(a_i, a_j)_{\substack{0 \leq i,j \leq n \\ i \neq j}}$ are not in ascending order. Such pairs are called *inversions*.

Remark.

This problem has numerous applications such as

- Voting theory
- Analysis of search engines ranking
- Collaborative filtering

Example

Given 6 movies compare the ranking of two users:

Movie	A	B	C	D	E	F
First user	1	2	3	4	5	6
Second user	1	3	5	2	4	6

Example

Given 6 movies compare the ranking of two users:

Movie	A	B	C	D	E	F
First user	1	2	3	4	5	6
Second user	1	3	5	2	4	6

Inversions: $(3, 2)$, $(5, 2)$, $(5, 4)$

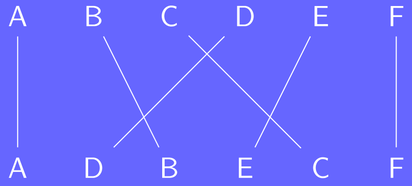
Example

Given 6 movies compare the ranking of two users:

Movie	A	B	C	D	E	F
First user	1	2	3	4	5	6
Second user	1	3	5	2	4	6

Inversions: $(3, 2)$, $(5, 2)$, $(5, 4)$

A simple geometrical view:



Divide and conquer approach

Strategy for solving the counting inversions problem (1.40):

1. **Divide:** split the list L into two halves L_1 and L_2
2. **Conquer:** recursively count inversions in each list
3. **Combine:** count inversions for the pairs (l_i, l_j) with l_i and l_j belonging to L_1 and L_2 respectively

The sum of the three counts is the total number of inversion in L

Example.

1	5	4	8	10	2	6	9	3	7

1	5	4	8	10

2	6	9	3	7

Divide and conquer approach

Strategy for solving the counting inversions problem (1.40):

1. **Divide:** split the list L into two halves L_1 and L_2
2. **Conquer:** recursively count inversions in each list
3. **Combine:** count inversions for the pairs (l_i, l_j) with l_i and l_j belonging to L_1 and L_2 respectively

The sum of the three counts is the total number of inversion in L

Example.

1 5 4 8 10 2 6 9 3 7

1 5 4 8 10   2 6 9 3 7

(5,4)   (6,3),(9,3),(9,7)

1 4 5 8 10   2 3 6 7 9

Divide and conquer approach

Strategy for solving the counting inversions problem (1.40):

1. **Divide:** split the list L into two halves L_1 and L_2
2. **Conquer:** recursively count inversions in each list
3. **Combine:** count inversions for the pairs (l_i, l_j) with l_i and l_j belonging to L_1 and L_2 respectively

The sum of the three counts is the total number of inversion in L

Example.

1	5	4	8	10	2	6	9	3	7

1	5	4	8	10

(5,4)

2	6	9	3	7

(6,3),(9,3),(9,7)

1	4	5	8	10

2	3	6	7	9

(4,2),(4,3),(5,2),(5,3),(8,2),(8,3),(8,6),(8,7),(10,2),(10,3),(10,6),(10,7),(10,9)

Merge and count

Algorithm. (*Merge and count*)

Input : Two sorted lists: $L_1 = (l_{1,1}, \cdots, l_{1,n_1})$ and $L_2 = (l_{2,1}, \cdots, l_{2,n_2})$

Output: The number of inversions *count*, and L_1 and L_2 merged into L

Function `MergeCount`(L_1, L_2)**:**

    $count \leftarrow 0$; $L \leftarrow \emptyset$; $i \leftarrow 1$; $j \leftarrow 1$;

    while $i \leq n_1$ **and** $j \leq n_2$ **do**

        if $l_{1,i} \leq l_{2,j}$ **then**

            append $l_{1,i}$ to L; $i{+}{+}$;

        else

            append $l_{2,j}$ to L; $count \leftarrow count + n_1 - i + 1$; $j{+}{+}$;

        end if

    end while

    if $i > n_1$ **then** append $l_{2,j}, \cdots, l_{2,n_2}$ to L;

    else append $l_{1,i}, \cdots, l_{1,n_1}$ to L;

    return *count* and L

Sort and count

Algorithm. (*Sort and count*)

Input : A list $L = (l_1, \cdots, l_n)$

Output: The number of inversions *count* and L sorted

Function `SortCount`(L)**:**

  if $n{=}1$ **then return** 0 and L;

  else

    Split L into $L_1 = (l_1, \cdots, l_{\lceil n/2 \rceil})$ and $L_2 = (l_{\lceil n/2 \rceil + 1}, \cdots, l_n)$;

    $count_1, L_1 \leftarrow$ `SortCount`(L_1);

    $count_2, L_2 \leftarrow$ `SortCount`(L_2);

    $count, L \leftarrow$`MergeCount`(L_1, L_2);

  end if

  $count \leftarrow count_1 + count_2 + count$;

  return *count* and L

end

Key points

- How to present pseudocode?
- What are the two main categories of data structure?
- What is a greedy algorithm?
- Describe the divide and conquer strategy
- How is the `Union-Find` data structure working?

Thank you!