Union Find

Data Structures and Algorithms
Andrei Bulatov

Union Find

In a nutshell, Kruskal's algorithm starts with a completely disjoint graph, and adds edges creating a graph with fewer and fewer connected components.

Every time an edge is added it must connect two different connected components.

We need a data structure that

(a) stores disjoint set of elements

(b) implements Make-Set procedure that creates a new element

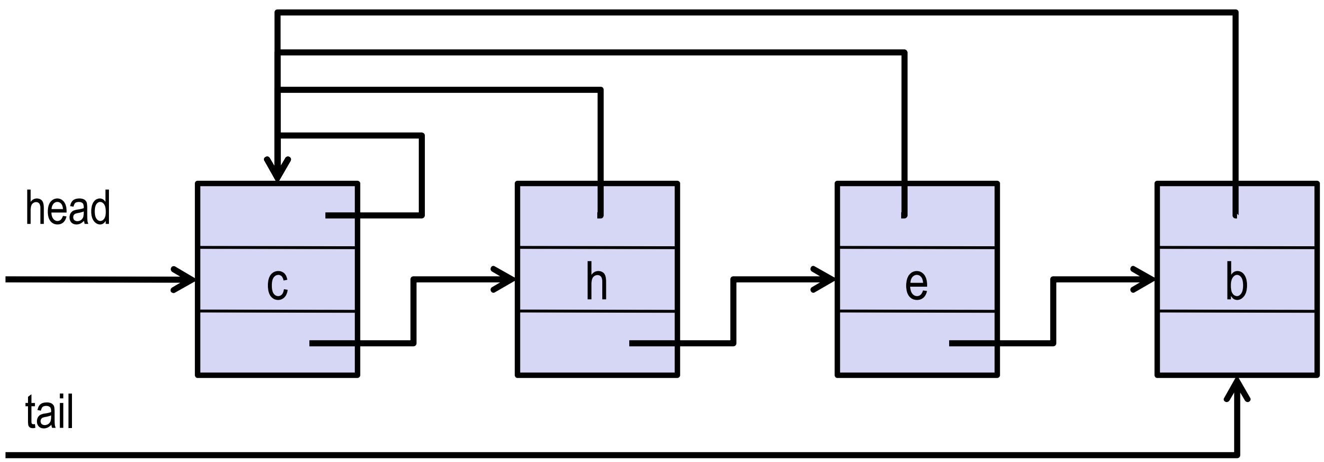
(c) implements the Find procedure that returns the name of the connected component containing its argument

(d) implements the Union procedure that given elements x and y merges the sets they are contained in into a single set

Union Find via Linked Lists

Every set represented by a linked list with extra pointer to the head

Also the head of the list is the name of the set.



Make-Set is easy

Find is easy: just return the pointer to the head

Union Find via Linked Lists (cntd)

Union(x,y):

Append x's list onto the end of y's list

Use y's tail pointer to find the place where to append x's list

Need to update the pointers to the head in x's list. It is linear in the length of x's list

If we merge sets of elements $x_1, x_2, \dots, x_n$ in the following order:

Union(x_1, x_2), Union(x_2, x_3), $\dots$, Union(x_{n-1}, x_n)

then the total number of pointers to update is

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

Weighted-Union Heuristic

The main slow down of Union is due to cases when a longer list is appended to a shorter list

Suppose that each list contains its length as extra piece of data, and we always append the shorter list to the longer one.

This is called weighted-union heuristic

Lemma

Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m Make-Set, Union and Find operations, n of which are Make-Set operations, takes $O(m + n \log n)$ time

Weighted-Union Heuristic

Proof

We compute for each object in the set of size n, the maximal number of times the pointer to the head of this object has been updated.

Fix object x

Every time the pointer of x is updated the elements comes from the smaller set

Therefore, after the first update, x belongs to a 2-element set

after the second update, x belongs to a 4-element set

• • •

Hence there are at most $\log n$ updates

Weighted-Union Heuristic

Proof

For n objects the maximal times pointers updated is $n \cdot \log n$

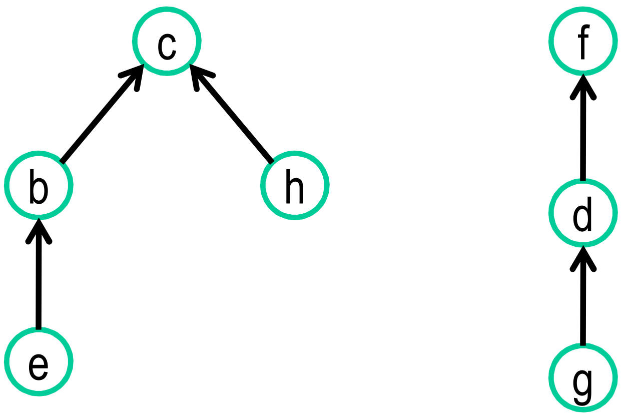
Every Make-Set and Find operation takes constant time

Thus the total time $O(m + n \cdot \log n)$

QED

Disjoint Sets Forests

Represent disjoint sets as forest



No better than through linked lists

Can be improved using heuristics

Disjoint Sets Forests: Heuristics

Union by rank:

Similar to weighted-union. Tree with fewer vertices points to the root of the tree with more vertices

Path compression

When performing a Find operation, we change pointers so that they point to the root

Lemma

Union rank alone gives O(m log n)

Path compression gives $\Theta(n + f \cdot (1 + \log_{2+f/n} n))$ where n is the number of Make-Set, and f the number of Find operations

Both O(m α(n)) where α(n) is a very slowly growing function

Divide and Conquer

Data Structures and Algorithms

Andrei Bulatov

Divide and Conquer, MergeSort

Recursive algorithms: Call themselves on subproblem

Divide and Conquer algorithms:

- Split a problem into subproblems (divide)
- Solve subproblems recursively (conquer)
- Combine solutions to subproblems (combine)

MergeSort

- Divide: Split a given sequence into halves
- Conquer: By calling itself sort the two halves
- Combine: Merge the two sorted arrays into one

Counting Inversions

Comparing two rankings

A ranking is a permutation of some objects

Objects can be numbered, and one of the rankings is just the natural order

The Counting Inversions Problem

Instance:

A permutation $a_1, \ldots, a_n$ of numbers 1, …, n

Objective:

Find the number of pairs i,j , i < j such that $a_i > a_j$

Algorithm Idea

Straightforward algorithm takes O(n^2) time

Use divide and conquer approach:

split the sequence into two halves

find the number of inversions in the halves

then what?

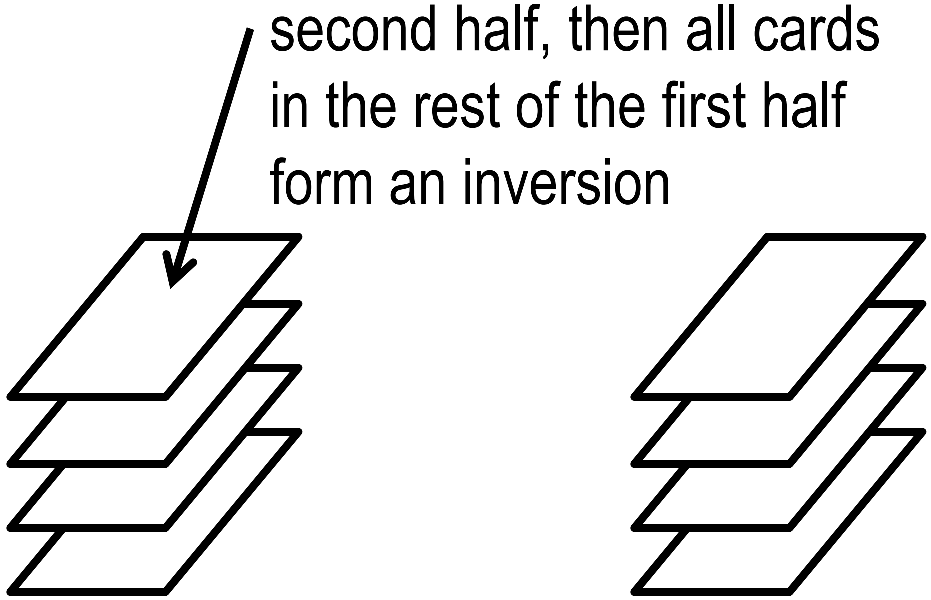
Observation:

`Between halves' inversion have the form (a_i, a_j) where a_i is in the first half, a_j is in the second half, and $a_i > a_j$

Algorithm Idea (cntd)

Assuming that the two halves are sorted we can run a procedure similar to Merge

If this card is greater that the one on the top of the second half, then all cards in the rest of the first half form an inversion



first half

second half

Algorithm

Merge-and-Count(A,B)

```
set curr1:=1, curr2:=1   /* current cards in halves
set count:=0             /* # of inversions
while curr1≠last1+1 and curr2≠last2+1
   if A[curr1]≤B[curr2] then do
      output A[curr1]
      set curr1:=curr1+1
   else do
      output A[curr2]
      set curr2:=curr2+1
      set count:=count+(last1-curr1+1)
   endif
endwhile
output the rest of the non-empty half and count
```

Algorithm (cntd)

Sort-and-Count(L)

```
If last=1 then no inversions
else do
   divide L into two halves:
     A contains the first ⌊last/2⌋ elements
     B contains remaining elements
   set (p,A):=Sort-and-Count(A)
   set (q,B):=Sort-and-Count(B)
   set (r,L):=Merge-and-Count(A,B)
endif
output p+q+r and the sorted list L
```

Sort-and-Count

Theorem

The Sort-and-Count algorithm correctly sorts the input and counts the number of inversions.

It runs in $O(n \log n)$ time for a list of n elements

Closest Pair: The Problem

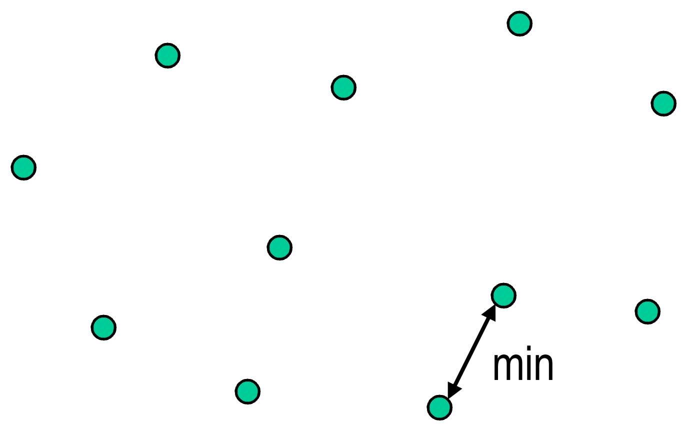
The Closest Pair Problem

Instance:

n points in the plane

Objective:

Find a pair of points that are closest together



Algorithm Idea

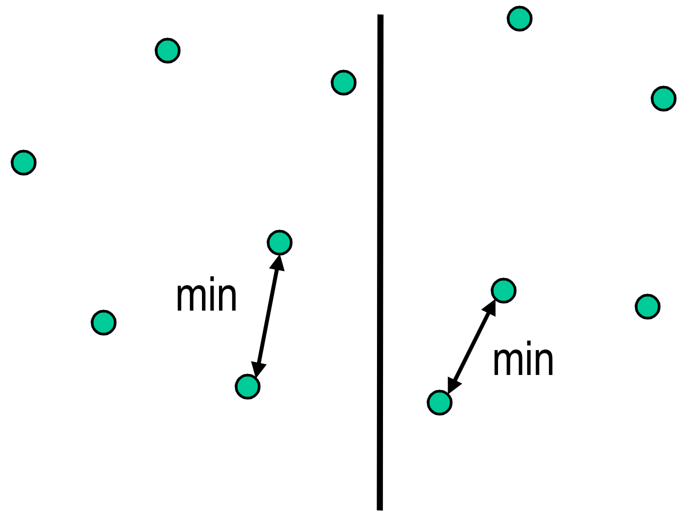
We have an algorithm that runs in O(n^2) time

Divide and conquer:

Split the set of points into two halves

Find closest pairs in the halves

Check if there is a closer pair crossing the border



Assumptions and Constructions

Assumption:

No two points have the same x-coordinate or the same y-coordinate

Let P be the set of points.

We create lists P_x and P_y of points sorted by the x- and y-coordinate, respectively

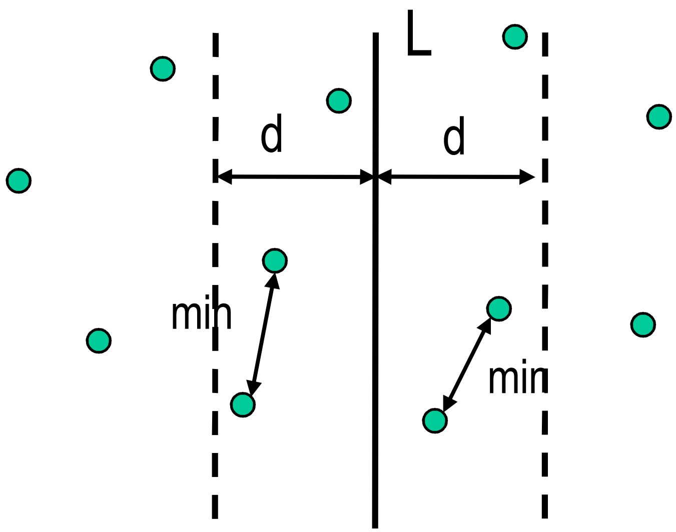
This can be done using the same divide and conquer process, sorting, and then merging the two halves.

Let Q and R be the two halves,

Q_x, Q_y and R_x, R_y the halves ordered with respect to the x- and y-coordinates

Let q_0^*, q_1^* and r_0^*, r_1^* be the closest pairs from the two halves

Crossing the Border



L is given by x = x*

Let δ be the minimum of $d(q_0^*, q_1^*)$ and $d(r_0^*, r_1^*)$

Lemma

If there exist $q \in Q$ and $r \in R$ such that $d(q,r) < \delta$, then each of q and r lies within a distance δ of L

Crossing the Border (cntd)

Proof

Suppose q and r exist, say, $q = (q_x, q_y)$ and $r = (r_x, r_y)$

We have $q_x \le x^* \le r_x$

Then

$$x^* - q_x \le r_x - q_x \le d(q,r) < \delta$$

and

$$r_x - x^* \le r_x - q_x \le d(q,r) < \delta$$

so each of q and r has an x-coordinate within δ of x* and hence lies within distance δ of the line L

QED

Crossing the Border (cntd)

Let S denote the set of points belonging to the band of width δ around L

Let also S_y be the set S sorted by increasing y-coordinate

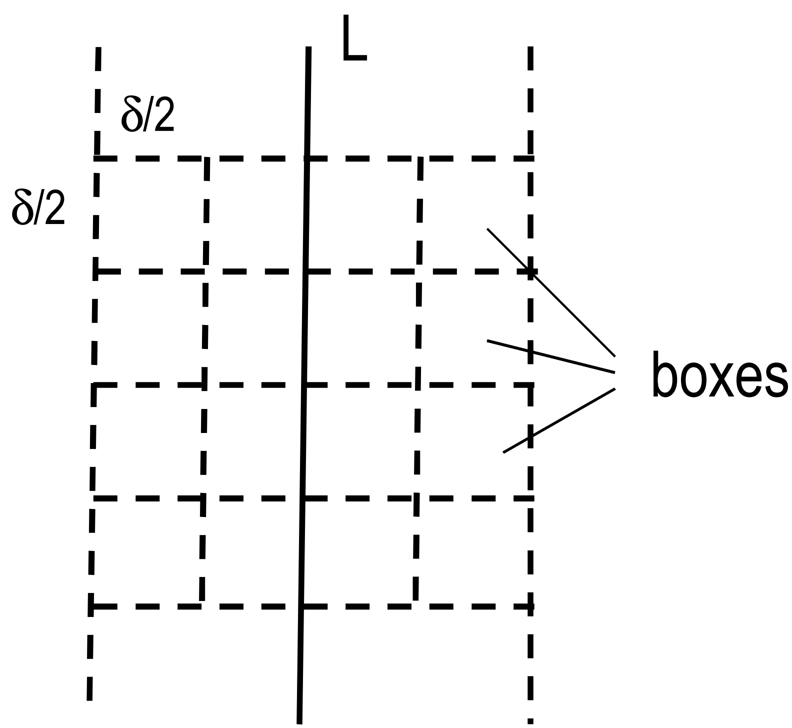
Lemma

There exist $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$ if and only if there exist $s,s' \in S$ for which $d(s,s') < \delta$

Crossing the Border (cntd)

Lemma

If s,s' $\in$ S have the property that d(s,s') < δ, then s and s' are within 15 positions of each other in the sorted list S_y



Crossing the Border (cntd)

Proof

Denote by Z the band of width δ around L

We partition Z into boxes: squares with side $\delta/2$

No two points belong to the same box, as it contradicts the assumption that minimal distance between two points on the same side of L is δ

Since the distance between s, s' is less then δ they also cannot be more than 2 boxes apart vertically

QED

Algorithm

```
Closest-Pair(P)
```

`construct` P_x `and` P_y `/*In O(n log n) time`

`set` (p_0^*, p_1^*)`:=Closest-Pair-Rec(`P_x, P_y`)`

Algorithm

```
Closest-Pair-Rec(P_x, P_y)
  if |P|≤3 then use brute force
  construct Q_x, Q_y, R_x, R_y            /*O(n) time
  (q0*, q1*):=Closest-Pair-Rec(Q_x, Q_y)
  (r0*, r1*):=Closest-Pair-Rec(R_x, R_y)
  set δ:= min{d(q0*, q1*), d(r0*, r1*)}
  set x*:=max x-coord. of points in Q,  L:={x=x*}
  set S:={points in P within distance δ from L}
  construct S_y
  for each s∈S compute dist. to next 15 points in
  if s,s' is the pair achieving the minimum and
   d(s,s')<δ return (s,s')
  else if d(q0*, q1*) < d(r0*, r1*) then return (q0*, q1*)
       else return (r0*, r1*)
```

Closest Pair: Analysis

Theorem

The Closest-Pair algorithm outputs a closest pair of points in P

Theorem

The Closest-Pair algorithm runs in $O(n \log n)$ time for a list of n elements

Integer Multiplication

How much time do we really need to multiply two numbers?

Standard algorithm

```
     1101
     1011
---------
     1101
    1101
  1101
---------
 10001111
```

takes O(n^2) time

Divide-and-Conquer Algorithm

Let x and y be given numbers, n digits each.

Represent them as: $x = x_1 \cdot 2^{n/2} + x_0, \; y = y_1 \cdot 2^{n/2} + y_0$

Then

$$xy = (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0)$$
$$= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0$$

We need to compute 4 smaller products

Does it help? Let T(n) be the running time

$$T(n) \le 4T(n/2) + Cn$$

By the Master Theorem

$$T(n) \le O(n^{\log 4}) = O(n^2)$$

Divide-and-Conquer Algorithm (cntd)

Reduce the number of recursive calls

$$xy = x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0$$

Compute $(x_1 + x_0)(y_1 + y_0)$ and then

$$x_1 y_0 + x_0 y_1 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0$$

Does it help?

Now we need 3 recursive calls: $(x_1 + x_0)(y_1 + y_0),\ x_1 y_1,\ x_0 y_0$

Thus $T(n) \le 3T(n/2) + Cn$

By the Master Theorem $T(n) \le O(n^{\log 3}) = O(n^{1.59})$