summary

33177 characters in 5133 words on 854 lines

Florian Moser

February 4, 2020

1 algorithm lab

overview
greedy with dp, sliding window, split & list
shortest paths with dijkstra, bellman-ford
minimal spanning trees with kruskal, prim
searches (binary, DFS, BFS)
matching with matching size
bipartite matching with cost, MaxIS, MinVC
connected components, strong components
max flow with min-cost, non-negative cost, min-cut
linear programming with non-negative
hit tests with min circle
delaunay triangulation with voronoi

ressources
https://algolab.m30m.ir/ for shared code
https://soi.ch/wiki/ for good wiki articles

2 libraries

2.1 standard library (STL)

default C++ library
sort uses lesser by default; pass ‘greater<int>()‘ to overwrite
‘operator <‘ overloads need to compare ‘self <other‘

2.2 boost

”academic” library with many fast algorithms

2.3 CGAL

geometric library

kernel
define how stuff is calculated
choose ‘exact_constructions‘ if constructions needed
choose ‘exact_constructions_with_sqrt‘ if square root needed
else choose ‘inexact_constructions‘

predicates
boolean function like ‘do_intersect‘
always exact (with the used kernels)
fast

constructions
function with return type ‘number‘ or ‘object‘
correctness depends on used kernel
slow to execute (hence avoid)

rounding
‘CGAL:to_double(exact)‘ may be inexact
hence compare result to original exact number and increase/decrease

building
‘cgal_create_cmake_script‘ && ‘cmake .‘ && ‘make‘

improve runtime
avoid constructions (try to get by with predicates)
avoid square roots (try to compare squared values)

3 implementation

containers
‘queue‘/‘stack‘ for only push/pop
‘deque‘ for front/back insertion
‘vector‘ for lookups; consider using ‘char‘, ‘short‘ instead of int
‘bitset‘ for boolean vectors
‘map‘ for complicated lookups
‘multiset‘ for min/max in sliding window. use ‘erase(find(condition))‘ to erase only single element.
‘priority_queue‘ for continuously sorted push/pop structure
‘disjoint_sets_with_storage‘ (union-find) for membership assignments
‘emplace_back‘ to construct directly into the container
avoid copies when iterating (const auto & element : element)

precision
10^n = 2ˆ(3.3∗n)
check multiplication (2b), addition (b+1), euclidian distance (2b+3)
int 2ˆ32, long 2ˆ64, double 2^{53} (mantissa width)
reformulate divisions to multiplication
ensure intermediate results similar magnitude than result

i/o
‘random_shuffle()‘ for malicious input
always include ‘fixed‘ and ‘setprecision(0)‘
use ‘\n‘ instead of ‘endl‘
‘ios_base:sync_with_stdio(false);‘ and ‘in.tie(0);‘

reduce constants
10^7 operations per second
reduce function calls
pass arguments to functions by reference (&)
for fixed string vocabulary, consider creating int mapping
use pointers ‘∗‘ with ‘new type()‘ if references needed

4 algorithms

4.1 searches

binary search (log n)
check if sorted range contains element
(loop while start ≤ end)
check element at index = start + (end-start / 2)
if element matches then abort
if element smaller then start = index+1
if element bigger then end = index-1

depth first search
visit all nodes depth first
pick start node and set to visited
recursively call for unvisited children

breadth first search
visit neighbours first
add start node to queue
pick from queue and add unvisited children to queue

4.2 utilities

check prime number (single lookup)
(loop while index++ ≤ sqrt(n))
check if n % index == 0

check prime number (Sieve of Eratosthenes)
create lookup(max_n, 0)
(loop while index++ ≤ max_n)
continue if number marked as non-prime
else add all multiplicies into lookup

GCD
recursively call GCD(B, A%B) until B = 0

others
longest common sequence
longest increasing sequence

5 problems

5.1 greedy

calculate single variation of problem

5.1.1 detect

10^7 input
able to pick optimal element at any time

5.1.2 proof arguments

exchange (would always choose better element)
staying ahead (no better strategy exists)

5.1.3 implement

preprocessing (usually sorting)
do greedy choice for all elements

5.1.4 algorithms

even pairs
see tutorial on slides
(value ∗(value - 1)) / 2 for (value over 2))

detect trees
n-1 edges and fully connected
exactly one path from leaf to root

5.1.5 problems (define)

even pairs
# of even ranges of size ≥2
see tutorial slides

even matrixes
# of even quadruples in matrix
use even pairs within summed up matrix columns

equal weight knapsack
choose elements with different value but same weights
sort by value/weight ratio ASC
choose until full

motorcycles
check if lines starting at x=0 with some slope intersect
sort by absolute slope ASC
remember most extreme point for positive / negative slope
if next element more extreme then OK

motorcycles subset
choose subset of lines which do not cross
sort by y coordinate ASC
pick longest increasing subset of slopes

minilength
minimize max edge length crossing network partition
sort edges DESC
pick until start/end in same component

missing roads
minimize cost of picked edges with only single edge per vertex choosable
sort by cost ASC
take if component has more edges than vertices

roads connecting
build roads within city looking like tree with no two roads to same city
start at leaf, select edge & remove parent node (and edge)
greedily continue until at top

chariot race (idea only)
select vertices with minimal weight such that all edges connect to selected vertice in tree
for an unselected vertex, select one of its children and tell the others the parent is unselected
for a selected vertex, tell its children its selected

5.2 sliding window

calculate best variation of the solution within a single pass

5.2.1 detect

10^7 input
can decide based on interval state if left/right increase
can update interval state based on left/right
(optional) need optimal interval

5.2.2 algorithms

count equal elements in lists
sort lists and start with $index_1 = index_2 = 0$
increase index of alphabetic lower entry

interval scheduling
sort by finish time ASC
select top & set min_start_time = top.start_time
skip while top.start_time <min_start

5.2.3 problems

deck of cards
range with certain sum
increase i or j to decrease/increase sum in between

search snippets
find shortest range with all words occuring in between
sort by position as tuple (position, word_id)
keep last known position for each word & min/max of current valid range (for each slide)
if min == last_position[word_id] then update min = min(other last_position)
if max == last_position[word_id] then max = position
last_position[word_id] = position
remember minimal diff max-min

beach bars
find segment with most parasols; also minimize max parasol distance from center of segment
have position counter which increments by one each round
adjust start/end index of included parasols if at start/end of segment matches
compare current_max_distance with max_distance
compare current_parasols_in_range with parasols_in_range

defensive line
maximize number of elements in segments with specific sum
do sliding window to find all segments with specific sum
safe in table with dp[start] → end
do DP over table choosing or not choosing some segment

attack of the clones
choose segments from circle (like intervals, but around circle)
transform to IR with (position, start|end_before)
find best starting position (fewest intervals active)
(loop for each active interval)
choose active interval
choose next with earliest finish time

boats
choose non-overlapping boats with width & attach point
add previous boat if none can end sooner anymore
start with left_index = -inf
if (left_index <attach_point) then add
else left_index = min(previous_left_index, current_left_index)

moving books
differently abled people carry differently weighted books
sort both people/weight DESC
find bottleneck (=max ratio) of people available/books liftable

falling dominos
check how many dominos of different sizes will fall
next_height = max(next_height, current_height)
–next_height when moving to next position

buddy selection
count shared interests of two buddies
sort both list of interests
scanline advancing the alphabetic lower one

evolution
answer many queries for some parent node within bounds
build up tree
do DFS while remembering path to root
query bounded ancestor rapidly with ‘lower_bound‘

new york
find range in tree of certain length and fulfilling min/max weight condition
build up tree
do DFS while remembering path to root, min/max with multiset
adapt min/max with multiset.erase(multiset.find(current_value))

5.3 bruteforce

calculate all variations of a problem and choose best result
add caching (”DP”) to speed up

5.3.1 detect

n^7 input
maximize / minimize something with optimal subproblems

5.3.2 DP

add caching vector to avoid computing same twice
ensure cache identifiers lead to acceptable cache size

5.3.3 split & list

gather solution for splitted problems
choose solution which does not contradict

5.3.4 implementation order

define return value
ensure leads to acceptable size of cache
define greedy decision (for example choose one & recurse over others)
ensure correct result calculated
use memoization (recursive) or bottom up (from known to unknown)
use split&list

5.3.5 problems

san francisco
need to beat given score with limited moves in game with different points per move
return points
bruteforce chosen canal until no more moves
cache #remaining_moves, max_archivable_score
(can not return moves for target score as caching impossible)

magician and the coin
maximize outcome to have target amount after coin flips with different probabilities
return probability of outcome
bruteforce optimal bet amount
cache round, available_amount
outcome = p∗recursive(won) + (1-p)recursive(lost)

fibonacci
calculate p = p-1 + p-2 for p0 = p1 = 1
return number
cache p's

metal rod cutting
maximize payoff with cuts of different lengths / price
return payoff
bruteforce chosen size
cache with size of uncut rod part

edit distance
minimize changes (insert, edit, remove) from one word to another
return changes needed
bruteforce insert/edit/remove to get from letter i to letter j
cache with position of words equal

burning coins
minimal payoff when each second coin can be chosen from the end of line
return payoff
bruteforce left/left, left/right, right/left, right/right take
cache with left-over coins dimension

from russia with love
coins in a row, n passengers take out coins at the end, maximize reachable score
return payoff
bruteforce left/right take with min if others, with max if self
cache with left-over coins dimension

roads connecting
build roads within city looking like tree with no two roads to same city
return max roads picked
bruteforce selected edge
cache with edge index & picked/not picked

light at the museum
switch flicks to change light configuration until target reached
return number of switches or INTMAX if impossible
bruteforce switch or not switch flik

light at the museum (split & list)
switch flicks to change light configuration until target reached
return valid solutions (which switches fliked)
bruteforce switch or not switch flik, no early outs
divide input in two & call recursive function for each to get step-solutions
chooes flik configuration which occurrs in both step-solutions

punch
minimize price maximize diversity of drinks at specific quantity & price
remember (cost, #drinks, selected_drinks);
go from left to right (bottom up); hence start with (0, 0, 0) at volume 0
increase volume by one, and update state for each drink
save state for that volume with lowest cost & most different drinks

5.4 shortest path

find shortest path between some start/end

5.4.1 detect

10^5 input
graph structure with needed shortest path

5.4.2 algorithms

'dijkstra_shortest_paths'
for nonnegative weights
can remember predecessor & distance map to source

'bellman_ford_shortest_paths'
also for negative weights
can remember predecessor & distance map to source

5.4.3 do-it-yourself

dijkstra (n logn + m)
take cheapest edge which connects to vertex outside network
create priority queue pq with (distance_from_start, vertex)
create distance map dm with [vertex] → min_distance_from_start
add first entry pq (0, start) and dm [start] → 0
(loop while !pq.empty)
select first entry with smaller/equal distance
relax dm entries for edges & add them to pq

5.4.4 modelling

prefer less vertices over less edges because O(v∗logv + e)

5.4.5 shortest paths problems

first steps with BGL
find longest path with 'max_element' on 'distance_map'
find edges on shortest path with 'dist[source] + weight == dist[target]'

ant challenge
use fastest of many different located/speedy routes
add all edges to graph
dijkstra deals with multiple same start/end edges

planet express
fastest route from multiple sources
add super-vertex with edges to all sources

bobs burden
minimize weight of chosen balls within triangle
get connections inside triangle from top-left to bottom-right
create edge between vertice in both direction with #ball_weight
run dijkstra from each edge node to get three dist_maps
pick node with min(sum_of dist_maps[node] - 2∗weight)

marathon
find all shortest path from source to sink
do dijkstra
remove edges were check dm[source] + length >dm[target]

5.5 minimal spanning tree (n logn)

connect vertices as cheaply as possible

5.5.1 detect

10^5
overall cheapest route inside network (not from specific start point)

5.5.2 algorithms

'kruskal_minimum_spanning_tree'
fast for most use cases

'prim_minimum_spanning_tree'
for heavily interconnected graphs

5.5.3 do-it-yourself

kruskal (m logm)
take cheapest edge without creating cycles
sort edges by weight ASC, create union find
(loop while !edges.empty)
pick edge with start/end in different sets
add edge to MST & union start/end

prim (m logn)
take cheapest edge which connects to vertex outside network

create priority queue with (distance_from_start, vertex)
add first entry (0, start)
(loop while !pq.empty)
select first entry leading to unvisited vertex
add edges from unvisited vertex to priority queue

5.5.4 problems

ant challenge
territory established by choosing shortest path to reach all trees
do MST with network of trees of species

return of the jedi
find best connections between planets different from optimal
do kruscal n times, while skipping different optimal choice each time

5.6 matching

assigning vertices or edges to one another

5.6.1 not greedy

maximal matching may not be maximum
which itself may not be perfect (no guarantee all matched)

5.6.2 detect

10^3 for O(mn)
assign vertex to other vertex
assign edge to two vertices

5.6.3 algorithms

'edmonds_maximum_cardinality_matching'(nm)
max subset of edges without sharing endpoints
'matching_size' afterwards to find matching size

5.6.4 problems

buddy selection
check if optimal buddy assignment possible
check if 'matching_size' == #buddies/2

consecutive constructions
max buildable roads without visiting city twice
create max matching between to-city and from-city nodes

satellites
install program on fewest ground station & satellites
find MaxIS

5.7 bipartite matching

assigning vertices to each other in bipartite graph

5.7.1 algorithms

bipartite matching
create graph with capa 1 edges
do max flow

bipartite cost matching
create graph with capa 1 edges & cost
do max flow min cost

maximum independent set (MaxIS)
maximal vertices so none connected by edge
do bipartite matching
mark reachable vertices in residual graph with BFS from source
choose labeled L and unlabeled R

minimum vertex cover (MinVC)
minimal vertices to reach all edges
size = #(edges in max matching)
like MaxIS but choose unlabeled L and labeled R

5.8 connected components (n+m)

find connected nodes

detect
10^7
"connected"

algorithms
'connected_components' to get mapping of vertex to component index

5.9 strong components (n+m)

find pairwise connected nodes

5.9.1 detect

10^7
"pairwise-reachable"

5.9.2 algorithms

'strong_components' to get mapping of vertex to component index

5.9.3 problems

universal vertices
all vertices that reach all others
find strong components
exclude strong component which is target of edge of other
output vertices of single remaining strong component

planet express
shortcuts which only works for marked, pairwise reachable vertices
create shortcut if vertex marked and inside same strong component

5.10 max flow

calculate flow from source to sink.

5.10.1 detect

<1000 vertices, <20000 edges
looks like LP but "easier" solvable

5.10.2 modelling

reduce vertices/edges
detect if it is possible with capacity 1 edges
collapse vertices with same meaning & sum their capacities
for example games with same players

multiple sources/sinks
create super source/sink with c = inf (same for sinks)

vertex capacity
create input/output vertex
connect with edge with c=vertex capacity

minimal capacity edge
create 'c_min' outflow on source
adjust edge capacity '-c_min'
create 'c_min' inflow on target

5.10.3 implement

'push_relabel_max_flow' (V^3)
fast, general purpose

'edmonds_karp_max_flow' (V*E*U)
for very sparse and low max capacity graphs

5.10.4 problems

edge disjoint paths
count paths which can not use same edge
source to network with all edge capacities = 1
possible endings of paths to super-sink with capa = inf
maxflow = amount of paths

circulation
if demand can be matched by supply
create super source & super sink
check if max-flow = demand

football game points
check if play outcome is archievable
source to game-layer with vertex per game with c = points per game
game-layer to team-layer with vertex per team with c = points if team wins
team-layer to sink with capacity = points needed to archive for outcome
check if max-flow = total archievable points

coin tossing
test if outcome of coin tosses archievable with only some tosses unknown
coin toss competition feasibility test
source to competitor-layer with vertex for each player vs player combination, c=#number of games
competitor-player to player-layer with c=#number of game
player-layer to sink with #points needed to archive
check if source flow = max flow
check if source to competitor-layer flow = player-layer to sink flow

shopping trip
visit streets at most once going to store
source to network with all edge capacities = 1

shop location endings to super-sink with capa = #shops
measure maxflow = amount of paths

tetris
build wall from blocks with given position & width without overlapping joints
create position-in & position-out node connected with capacity = 1 (for joints)
insert blocks from start-position-out to end-position-in
measure max-flow from source (0-in) to sink (n-position-in)

london
snippets with letters on front/back to construct a message
source to snippet-layer for each snippet combination with capacity = #snippets
snippet-layer to letter buildable with #snippet
letter-layer to sink with #letter_required
check max-flow = sum_{of} (#letter_required)

london
snippets with letters on front/back to construct a message
source to letter with #front
connect letter with other letter for #back
connect letter to sink with #required_letters
check max-flow = sum_{of} (#letter_required)

marathon
how much capacity on shortest path from source to sink
use dijkstra to remove edges not on shortest path
do max flow on resulting network

surveillance photographs
go from red nodes to blue nodes; get back to red witout using edges twice
two networks, second one with all edges c = 1
connect source to red nodes in network1 with capacity = #red_weight
connect network1 blue to network2 blue with c = #blue_weight
connect network2 red to sink with c = #red_weight
measure max-flow

phantom menance
how many spaceships to block all routes at planets
create in/out vertex for planets with capa = 1
connect out-vertices to in-vertices if route with capa = 1
measure max flow

5.11 max flow min cost

calculate flow from source to sink while minimizing cost.

5.11.1 dimensions

up to 1000 nodes

5.11.2 modelling

use whole-number weights
adjust negative weights
reduce vertices & edges

5.11.3 adjust negative weights

add max-reward to all reward edges if visited only once
add constant factor per unit (time unit, distance unit) to all edges

5.11.4 algorithms

'successive_shortest_path_nonnegative_weights' (m^3)
for non-negaitve weights (up to 1000 edges)

'cycle_canceling' (C*nm)
for negative weights (up to 600 edges)

5.11.5 problems

canteen
make, sell & store menus for different prices & quantities
set max_earnings to max profitable menu
source to day-layer with producable quantity and cost = production cost
day to next day with storable quantity and cost = storage cost
day-layer to sink with sellable quantity and cost = max_earnings - sell price
check if max_flow = sum_of(sellable quantities) else abort
reward = sum_of(sellable quantities) *max_earnings - flow_cost

india
transport suitcases between cities for different prices/capacities with max cost
build normal max_flow min_cost network
add super_source with some set capacity
binary search source capacity until under allowed max_cost

carsharing
choose trips with start/end/start_time/end_time/reward to maximize profit
set max_earnings for max reward by time unit
car-station-layer with node per start_time/end_time
connect nodes with next lower with capa = #inf and cost = time_diff *max_earnings
car-station-layers to each other for each trip with capa = 1 and cost = time_diff *max_earnings - reward
source to first each first car-station-layer node with capa = initial_capacity and cost = time_diff*max_earnings
each last car-station-layer to sink with capa = inf and cost = time_diff *max_earnings
reward = sum_of(initial_capacity) *max_reward *total_time_units - flow_cost

tour of gaul
transport goods with different start/end/reward on route with different constrained capacities
set max_earnings to max reward some good earns
source to trip starts with #capacity_next_trip with cost = 0
trip starts to trip ends with #capacity_next_trip with cost = max_earnings*capacity
trip ends to sink with #capacity_next_trip and cost = 0
good start to good end with capacity = 1 and cost = distance*max_earnings - reward
reward = total_capacity *max_earnings - flow_cost

missing roads
minimize cost of building roads while only one road can be build per city
source to city-layer with capacity = 1 and cost = 0
city-layer to roads layer with one node per road, connected if road has ending in city
set capacity = 1 and cost = cost of building street
connect street-layer to sink with capa = 1 and cost = 0
cost = flow_cost and built roads = flow

bobs burden
in triange of balls choose one connected to edges to minimize weight of connected
create in/out vertex for each ball connected by edge with capa = 1, cost = weight
connect out vertex with in-vertex if reachable with capa = 1, cost = 0
do for each ball=source, edges = sink max-flow min-cost

5.12 minimum cut

find the bottleneck in max flow

5.12.1 implement

do BFS on maxflow to reachable (reverse capacity >0) vertices

5.12.2 problems

cantonal courier
buy priced zones to fulfill payed jobs
source to zone-layer with price as capacity
zones to jobs with capa = inf
jobs to sink with capa = reward
min-cut at jobs not taken (because less money for jobs than for needed zones)
min cut at zones bought (because less money for zones than for jobs connected)
reward = sum_of(job rewards) - flow

algocoön
minimal cost to cut directed & weighted graph in half
find minimal flow_cost for each pair (i, i+1)
output min_{cut} of that flow
works because minimal cut must be between some (i, i+1)

5.13 linear programming (LP)

minimize objective functions subject to constraints

5.13.1 solution variants

optimal (feasible, minimal solution)
unbounded (unlimited feasible solutions)
infeasible (no feasible solution)

5.13.2 detect

few (<200), linear input (constraints) or output (variables)

1000 constraints OK if very few variables & vice versa

5.13.3 modelling

maximization
invert variables, invert objetive-value

inside halfplane
a^t∗x <b for a^t description of halfplane
for point x,y this translates to a0x + a1y <b
variables x, y
constraints with a0x + a1y <b

position polynomial between points
degree is size of multiplicities from 0..d
for d = 1 include ax, by, c
for d = 2 include ax, bx^2, cxy, cy^2, dy, e
variables x, y, b
constraints with ax + by + 1c = 0

distance to line
a^t∗x = b for a^t description halfplane
$||a||_2$ = sqrt(a0∗a0 + a1∗a1) = sqrt(a^t ∗a) denotes norm of vector a
unit_distance = a/$||a||_2$ is normalized distance in direction of a
x = p + d∗unit_distance for x point with distance d from p
fill in x into halfplane description to get distance from halfplane
variables x, y, d
constraints with a0∗x + a1∗y + unit_distance∗d = b

avoid cross line
calculate positivity of a^t∗x + c >0 then >else <
variables x, y, c
constraints with a0∗x + a1∗y + c with same relation

non-vertical line
constraints of form a∗x + b∗y + c = 0
lower bound of a $\geq$ 1; for non-horizontal for b $\geq$ 1

5.13.4 implement

construct
'int', 'long', 'double' as trivial input types
'Gmpz', 'Gmpq' as input type if needed; construct late like 'lp.set_a(j, i, 1 / IT(number));'
'Gmpz' as output type; or 'Gmpq' if this is input type

fill
'lp.set_a(column, row, value)' to set entry in matrix
'lp.set_b(row, value)' to set entry in b
'lp.set_r' to set relation (prefer to reformulation of coefficients)
'lp.set_l(column, true, number)' for lower bound
'lp.set_u(column, true, number)' for upper bound
'lp.set_c(column, value)' for objective; '-value' if maximization
'lp.set_c0(value)' for offset

run
'options.set_pricing_strategy (CGAL:QP_BLAND)' if prone to cycling (infinite loops)
'options.set_pricing_strategy (CGAL:QP_PARTIAL_FILTERED_DANTZIG)' with 'double' IT if diff variables/constraints high
'solve_linear_program(lp)' for the general case
'solve_nonnegative_linear_program(lp)' if variables positive & unbounded

result
's.is_valid()', 's.is_unbounded()', 's.is_infeasible()' to check solution
's.objective_value()' to get result; if maximization multiply by '-1'
avoid post-processing with 'objective_value' to avoid rounding errors
do not copy result (known bug)

5.13.5 problems

separate red/blue points with line
variables x, y, c (resulting variable)
blue constraints as ax + by + c $\geq$ 0 for point (a,b)
red constraints as ax + by + c $\leq$ 0 for point (a,b)

diet
nutrition/price by food, minimize cost under min/max nutrition constraints
variables as units of food
objective function as price
constraint per nutrition min and max
minimal_cost = objective_value()

inball
maximize circle radius within lines for d dimensions; lines given as halfplanes a^t∗x <b for any point x
model as any middle point which maximizes the distance to all lines
variables as coordinates middle point (x) and radius (r)
objective function as -r (middle point objective is 0 because position irrelevant)
constraints as a∗x + r∗$||a||_2$ <b
radius = -objective_value()

radiation
minimze dimensions of polynomial dividing points
do binary search to find minimal feasible d polynomial

suez
rectangles at assigned poisiton & fixed aspect ratio, some with predefined size; maxmize circumference of others without overlapping.
rectangles touch overlap if x0+ratio/aspect >x1 - ratio/aspect && y0+ratio/aspect >y1 - ratio/aspect
variables are expand ratio per variable size rectangle
objective function -2(aspect_width + aspect_height)
constrain variable size rectangle with x1 + x2 <max((x0-x1)/aspect, (y0-y1)/aspect)
upper limit expand ratio by checking for each variable with each fixed size poster
total max circumference = -objective_value()

the empire strikes back
choose radius & weight of circles; must reach red points with certain weight, must not reach blue points. minimize total weight needed.
variables are weight per circle
objective function 1
constraints are min weight by red point; radius determined if closer than blue points
total weight = objective_value()

motorcycles starttime
start at x=0 with some slope with chosen starting time. minimize time between intersection of tracks.
variables are starting time and frustration tolerance (time between track intersection)
constraint for each intersecting line, $start_1$ + ||b1 - q||<$start_2$ + ||b2 - q||+ f
inverse constraint than above (swap start_1, b1 with start_2, b2) to enforce in both directions
objective function 1f
frustration tolerance = objective_value()

worldcup
sources with supply/concentration to targets with demand/max_concentration. maximize revenue
variables for each source / target variation
objective function -revenue by source/target combination
constrain for source that supply by target lower than supply
constrain for target that supply by source equal demand
constrain for target that concentration by source lower than max_concentration
profit = -objective_value()

roman lines
choose point to maximize weighted distance to lines without crossing lines to given point (x0, y0)
relation = by a0∗x0 + a1∗y0 + c <0 ? 1 : -1
norm = sqrt(a0∗a0 + a1∗a1)
variables for x, y, d
objective function -d
constrain variables by a0∗x + a1∗y + weight ∗norm ∗relation ∗d <-c (or >if relation == -1)
result = -objective_value()

5.14 hit test

need to hit some object with each other

hit
check if ray hits any segment
'do_intersect' with ray and each segment
early out after first success

first hit
get closes intersection point of ray to segment
'do_intersect' with ray and segment until first success
construct segment with 'intersection' and start point of ray
'do_intersect' with that segment and the rest
(hence intersection point only constructed if closer than before)
randomize order to avoid malicius inputs

5.15 minimal circle

minimal circle to include all points

5.15.1 problems

antenna
connect all people in area via antenna
use ‘min_circle‘

almost antenna
connect all people except one via antenna
use ‘min_circle‘ to find ‘circle‘
do 3 times ‘min_circle‘ without one of the ‘circle.support_points_begin()‘
early out if more than 3 support points

5.16 delaunay triangulation / voronoi (n log n)

get triangulation of points with smallest angles
composed out of maximal empty disks

5.16.1 maximal empty disk

disk which can not be expanded anymore & no points inside
construct inclusion-maximal disk (two points on border, none inside)
then move center & expand until third point hit

5.16.2 voronoi dual

can be easily constructed from delaunay
defines regions with the same closest point
on edges there are two closest points, on vertices there are ≥ 3

5.16.3 properties

maximizes smallest angles (no thin triangles)
contains all minimum spanning trees
nearest neighbour graph
3n-6 edges, 2n-4 faces

5.16.4 implement

data structure
‘Triangulation_face_base_2_with_info‘ if payload per face needed
‘Triangulation_vertex_base_2_with_info‘ if payload per vertex needed
‘Triangulation_hierarchy_2‘ if many point location queries needed

explore
‘t.insert(vector<Point>())‘ so CGAL can choose a good ordering (speedup of 30%!)
‘t.insert(vector<pair<Point, int>>())‘ to prefill ‘info()‘ of vertices
‘t.finite_vertices_begin()‘ for vertices except infinite vertex (in ‘all_vertices_begin()‘)
‘t.finite_faces_begin()‘ for faces inside the convex hull (in ‘all_faces_begin()‘)
‘t.finite_edges_begin()‘ for edges except infinite edges (in ‘all_edges_begin()‘)
‘t.is_infinite(edge_handle|face_handle|vertex_handle)‘ to check if infinite

navigate
‘t.locate(Point)‘ for face of point
‘t.incident_vertices(vertex_handle)‘ for adjacent vertices
‘t.incident_faces(vertex_handle)‘ for adjacent faces
‘t.incident_edges(vertex_handle)‘ for adjacent edges
‘f.neighbour(cw(i))‘ for vertex left, ‘f.neighbour(cww(i))‘ for vertex right

voronoi
‘t.nearest_vertex(Point)‘ for nearest vertex of point
‘t.dual(face_handle)‘ for center of circle used to construct face

construct
‘t.segment(face_handle, i)‘ for segment to neighbour i
‘t.segment(edge_handle)‘ for segment from edge

examples
‘t.incident_vertices(t.infinite_vertex());‘ to get convex hull
‘edge→first→vertex(trg.cw(edge→second));‘ to get the vertex at the end of an edge

5.16.5 problems

bistro
find nearest bistro to new location
simply query ‘nearest_vertex‘

graypes
find distance between closest points in some point cloud
iterate over edges of delauney and pick shortest one

h1n1
excape under minimum distance per query from crowd of people
use ‘with_info‘ for faces to index them properly
starting from infinite faces inwards remember max distance to outside (”dijkstra style”)
for each query, ‘locate‘ the face and check if distance safe

goldeneye
get min radius needed to move from source → target under circles from given points.
extract all edges in delaunay sorted by length
get ‘nearest_vertex‘ of source/target v1, v2
union start/end of edges until v1 & v2 in same set (union-structure)
radius = min(max(last_included_edge, source→v1, v2→target), v1→v2)

light the stage
output queries within given radius of ordered points where no more untouched queries remain
check with ‘nearest_vertex‘ if any point close enough; else early out
check if any query is close enough; else early out
check with ordered lamps where hit
output longest surveivors

clues
two color circles & ensure no center point within circle of other same color and ensure queries connected
do BFS on delauney
create new component, follow short edges (BFS) and ensure they are of different color
do triangulation per vertex color and ensure all edges are too long
query by checking if reachable & component matches

revenge of the sith
check max visitable points while with each visit some additional ordered point is blocked
binary search m; amount of planets visitable
do delauney with points except those already blocked
create graph from delauney with visitable edges
check if some component of size m exists (‘connected_component‘)

minidist
minimize distance of points to line which must divide two specific points
do convext hull
(for each parallel to one of convex hull edges)
check most optimal line; could be in the middle or if invalid on one of the two points

germs
check percentage of circles with expanding radius touching others or border
do min(distance to border, nearest germ from delauney)
sort min then query [0] (first touch), [n/2] (50% touch), [n-1] (100% touch)

worldcup
cost when for each circle traversed from source to target; only few points inside circle
look with delauney if inside circle else skip
check with all other points if distance to center both smaller or both larger than radius
else cost occurrs

motorcycles radio
minimal reach of radio station to cover motocycle path segment
traverse edges of voronoi and cut with segment
remember max distance of these cuts to its respectively closest radio station.