CEOI 2006 - Solutions

CEOI 2006 Scientific committee

Abstract

Contained herein are solutions for the problems featured at the 13th Central European Olympiad in Informatics which took place in Vrsar, Croatia in July 2006. Solutions are given as high level descriptions of the algorithms accompanied by pseudo-code.

1 Antenna

A brute force solution for this problem can be derived from the fact that a circle with the smallest radius always either passes through some three points or has a diameter that is a line segment connecting some two points. A brute force solution iterates through all such circles and for each circle counts the number of points contained. Only simple analytic geometry is required in this solution – calculating the center and the radius of a circle given three points comes down to solving a system of two linear equations. Clearly, this solution works in time $O(N^4)$.

One way to obtain more efficient solutions is by solving a simpler problem – given a number R find a circle with radius R containing at least K points (or determine that no such circles exist). If there is an efficient solution for this problem, we can solve the original problem using binary search – we find the minimal R (up to the required precision) for which such a circle exists. In general, a $O(f(N))$ solution for the simpler problem gives us a $O(If(N))$ solution for the original problem, where I is the number of iterations required for the given precision. With the given limits, I has to be around 30.

Suppose we are given the radius R and just one point A. There are many circles with radius R touching A, however, using a sweeping algorithm we can calculate the number of points contained in each of them in one pass. Imagine any circle of radius R passing through the point A. As we rotate the circle around A (with the radius fixed) two kinds of events can happen; a point may enter the circle, or a point may exit a circle. If we could sort all of these events in time and process them in order, we would know the total number of points contained in the circle at each moment.

Assume that A is the origin and that we start with a circle centered at $C = (R, 0)$. As we rotate the circle counterclockwise, each position can be represented by an angle between the segment $\overline{CA}$ and the x-axis. Given a different point B, its enter and exit angles can be computed using simple trigonometry. The solution calculates all the enter and exit angles, sorts the events and scans them keeping track of the number of points currently inside the circle. One implementation issue is how to handle the points already inside the circle in the

starting position. Solution given here assumes that the circle is initially empty and makes two full rotations to account for these points.

Algorithm 1 Antenna, given R find a circle of radius R containing K points

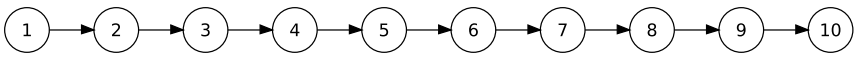
```
1:  for all points A do
2:    {Find all events}
3:    for all points B different from A such that d(A, B) ≤ 2R do
4:      α ← atan2(y_B − y_A, x_B − x_A)
5:      β ← acos(d(A, B)/(2R))
6:      Events.add(α − β, "Entering")
7:      Events.add(α + β, "Exiting")
8:      Events.add(α − β + 2π, "Entering")
9:      Events.add(α + β + 2π, "Exiting")
10:   end for
11:   Sort the events, lower angle comes first
12:   {Sweep}
13:   count ← 1
14:   for all events e do
15:     increment or decrement count based on the type of the event e
16:     if count ≥ k then
17:       We have found our circle
18:     end if
19:   end for
20: end for
21: There is no such circle
```

It is easy to see that the complexity of the suggested solution is dominated by the sorting routine inside the loop. If an efficient sorting scheme is used then the complete solution works in time $O(IN^2 \log(N))$.

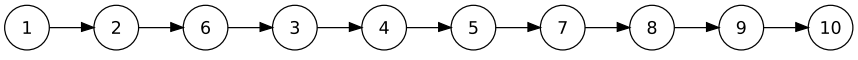
Also, due to the geometric nature of the problem, it was possible to achieve near perfect score with a heuristic solution or with a well optimized solution with a higher asymptotic complexity.

2 Queue

We start with a long linked list and perform a sequence of operations of the type *move node A in front of node B*. For example, the first part of the list looks like:



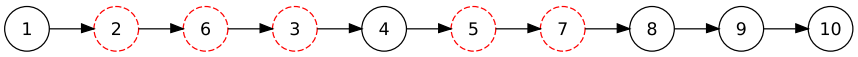
Moving node 6 in front of node 3 makes the list look like:



To solve the problem we need to realize that even after many operations, many nodes will retain their starting predecessor and successor. We will call

node X *trivial* if it is preceded by node $X-1$ and succeeded by node $X+1$. Nodes for which this property was violated at any point will be called *nontrivial*. In the beginning, all nodes are trivial.

Each move operation makes at most 5 previously trivial nodes nontrivial. In the previous example, nodes 2, 3, 5, 6 and 7 become nontrivial as a result of the move operation:



A limit of 50000 operations means that there will be at most 250000 nontrivial nodes in the end ($O(N)$ in any case). This number is small enough that we can keep track of them - for each nontrivial node we need to know its predecessor and successor.

Assume that we have an appropriate set data structure in which to hold the nontrivial nodes. The following algorithm processes the move operations. When *pred* or *succ* is referenced for a node, it implies finding whether the node is nontrivial and returning as appropriate. When *pred* or *succ* is assigned a value, it implies making the node nontrivial (if it's still trivial) and inserting or updating it in the data structure.

Algorithm 2 Queue, process move operations

```
for all operations "move node A in front of node B" do
    succ[pred[A]] ← succ[A]
    pred[succ[A]] ← pred[A]
    succ[pred[B]] ← A
    pred[A] ← pred[B]
    succ[A] ← B
    pred[B] ← A
end for
```

After the above snippet finishes we sort the questions so that we can efficiently find all questions between some two numbers (using binary search). A single pass through the long (compressed) linked list allows us to answer all questions.

We still have to choose the data structure in which to hold the nontrivial nodes. The operations we need are:

1. find node with given label
2. insert node with given label
3. find first node with label larger than given label

A binary search tree with some sort of balancing will suffice. The time complexity of processing the operations is $O(N \log N)$. The complexity of the list traversal is $O(N \log N + N \log Q + Q)$.

An additional bit of insight reveals that all nontrivial nodes appear as $A-1, A, A+1, B-1$ or B in at least one operation. A preprocessing pass over the operations allows us to find the nontrivial nodes and store them in a sorted array which, armed with binary search, acts as a replacement for the more complicated binary search tree structure.

Algorithm 3 Queue, answer all questions

```
1: Sort P-type and L-type questions
2: pos ← 1
3: label ← label of first node
4: while we haven't processed all nontrivial nodes do
5:   if succ[label] is nontrivial then
6:     Answer P-type questions for X = label
7:     Answer L-type questions for X = pos
8:     pos ← pos + 1
9:     label ← succ[label]
10:  else
11:    next ← first nontrivial node after label
12:    {nodes between label and next are in positions between pos and pos +
       next − label}
13:    Answer P-type questions for X ∈ [label, next − 1]
14:    Answer L-type questions for X ∈ [pos, pos + next − label − 1]
15:    pos ← pos + next − label
16:    label ← next
17:  end if
18: end while
```

3 Walk

Let P be any path from the destination point D to the starting point S. We will say that P is a *special path* if in each step it goes left as far as possible i.e. each segment in P goes left until it hits an obstacle or reaches the y axis and then goes either up or down until it can go left again (or until it reaches the starting point). We will argue that we only need to consider special paths i.e. there is always a shortest path from D to S that is special. This is somewhat intuitive, we can go around each building and the point S is strictly to the left of all buildings so it cannot hurt to consider only special paths. The formal proof of this claim is not trivial and not needed in order to solve the problem, but for the sake of completeness, we sketch the argument here.

Take any shortest path P from the point D to the point S. Consider the first point A after which P is not special, i.e. P can go left after A but instead, it goes either down or up. Let Q be a *left-down* path from A, i.e. Q starts at A and at each step goes left if possible, and down otherwise. Let Q' be a *left-up* path from A defined in a similar fashion. Note that both paths Q and Q' are special. Path Q will intersect the y axis at some point B, similarly Q' will intersect the y axis at some point B'.

- If S is below B then the path Q from A to B (denoted by Q_{AB}) followed by the segment BS is a shortest path from A to S, together with the path P_{DA} it forms a shortest path from D to S that is special.
- If S is above B' then the reasoning is similar.
- If S is between B and B' then P had to intersect Q or Q' at some point after A. Assume that P intersects Q at a point A' after A. Now a path $P_{DA} + Q_{AA'} + P_{A'S}$ is still a shortest path and is special from D to A'.

By repeating this step some number of times we get a special path from D to S.

Now we can turn our attention to the algorithm. For each building T we will calculate the shortest distance from S to the upper-right corner T_U and the lower-right corner T_D of T. Also, for each corner point A we will say that the T is the *next building from* A, denoted $T = next(A)$ if going left from A we first encounter T. If there is no such building we write $next(A) = \varepsilon$. For a corner point A, let $s(A)$ be the length of the shortest path from the starting point S to the point A, and let $d(A, B)$ be the Manhattan distance between the two points. Since we are only considering special paths the following holds.

$$s(A) = \min(s(T'_U) + d(T'_U, A), s(T'_D) + d(T'_D, A)) \text{ where } T' = next(A)$$

This recursive relation gives rise to the following algorithm.

Algorithm 4 Walk, find the shortest distance from S to every corner

```
1: Sort the rectangles by the larger x coordinate in increasing order
2: for a rectangle T, A = T_U, T_D do
3:    if next(A) = ε then {we have a trivial path from S}
4:       s(A) ← d(A, S)
5:    else
6:       R ← next(A)
7:       s(A) ← min(s(R_U) + d(A, R_U), s(R_D) + d(A, R_D))
8:    end if
9: end for
```

If we include a dummy rectangle T' with D in its upper-right corner, then the length of the shortest path from S to D is simply $s(T'_U)$. The actual shortest path can be easily reconstructed if, for each corner point A, we remember which of the two values in the recursive relation is lower.

We still need to show how to compute values $next(A)$ for each upper-right and lower-right corner A. First notice that we only need to consider right side of every rectangle. For a rectangle T, let $T.x$ be the x coordinate of the right side and $T.y_0$ and $T.y_1$ the y coordinates of the lower and the upper side. Clearly T_D has coordinates $(T.x, T.y_1 - 1)$, while T_U has coordinates $(T.x, T.y_2 + 1)$. Suppose we scan all the rectangles with an upwards moving line parallel to the x axis. We distinguish three kinds of events:

1. The line encounters the bottom of the rectangle T ($T.y_1$), we *activate* the rectangle T at the coordinate $T.x$.

2. The line encounters the top of the rectangle T ($T.y_2$), we *deactivate* the rectangle T at the coordinate $T.x$.

3. The line encounters T_D or T_U, we can find $next(T_D)$ or $next(T_U)$ as the rightmost active rectangle left from $T.x$.

We will use a variant of the *tournament tree* data structure to keep track of the active rectangles and answer queries. This data structure contains an array of $[0, \ldots, M]$ elements, where M is the largest of all x coordinates. Initially, all values of the array are set to ε. When a rectangle T is activated, the value T is

stored at an index $T.x$ in the tournament tree. The tournament tree structure usually allows us to find the smallest or largest value on a contiguous part of the array. Instead of finding a maximal value on the interval our variant finds a right-most value in the interval that is different from ε. Since rectangles do not intersect we can have at most one active rectangle at any given point x.

Algorithm 5 Walk, computing the values of *next*

```
1: {Find all events}
2: for all rectangles T do
3:    Events.add(T.y_1, "Activate rectangle T")
4:    Events.add(T.y_2, "Deactivate rectangle T")
5:    Events.add(T.y_1 − 1, "Query T_D")
6:    Events.add(T.y_2 + 1, "Query T_U")
7: end for
8: Sort all events by the y coordinate increasingly
9: {Sweep}
10: for all events e do
11:    if e = "Activate rectangle T" then
12:       Tournament.set(T.x, T)
13:    else if e = "Deactivate rectangle T" then
14:       Tournament.set(T.x, ε)
15:    else if e = "Query T_D" then
16:       next(T_D) ← Tournament.get(0, T.x)
17:    else if e = "Query T_U" then
18:       next(T_U) ← Tournament.get(0, T.x)
19:    end if
20: end for
```

Since each tournament tree operation takes $O(\log M)$ time, the total time complexity of the solution is $O(N \log N + N \log M)$.

4 Connect

We will use the term *extended room* to denote a set of positions on the board consisting of one room and the four neighboring corridors. It is not hard to see that if the game has been played correctly then every extended room on the board will be in one of the eleven possible configurations (ignoring the blocked corridors) given in Figure 1.

Conversely, suppose we have placed the dots on the board in such a way that each extended room is in one of the given configurations. Every figure will be connected to exactly one other figure – a path must lead out of every figure (configurations 2–5), and a path may not end except at a figure (configurations 6–11). It is possible that we have constructed a number of loops – closed paths not connected to any figure. However, if we find a placement of dots satisfying the above criteria which, additionally, uses the *smallest possible* number of dots then we have clearly found a solution.

Now, we can use dynamic programming to solve the problem. We will process rooms column by column, left to right, and each column top to bottom. For every room we will try all possible configurations consistent with the dots placed

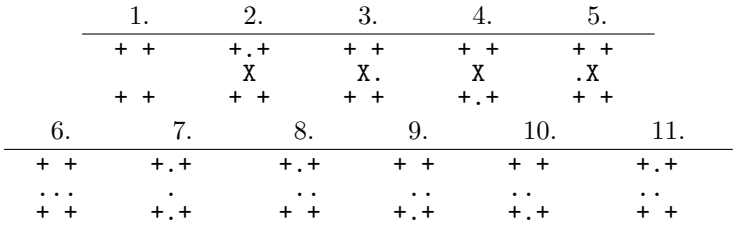


Figure 1: Connect, all allowable configurations

so far and the positions of figures and barriers. Clearly, we only need to keep track of the state of the corridors adjacent to the rooms we have not considered yet.

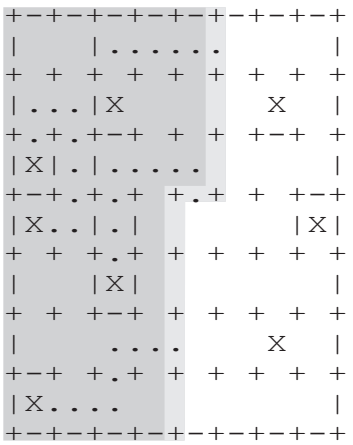


Figure 2: Connect, an example state

Each state can be described by the coordinates of the current room and a bit-array describing the state of the corridors between processed rooms and those yet to be processed (with the value 1 for a corridor with a dot, and 0 for an empty or blocked corridor). For example, the state given in Figure 2 can be described by the triplet $(4, 5, 10010010)$. The algorithm processes the room by considering all possible configurations consistent with the current state and the layout of the board. For example, in the current state we can continue with configurations 7 and 8 resulting in states $(5, 5, 10001010)$ and $(5, 5, 10010010)$ respectively. For a state S, let $B(S)$ be the smallest number of additional dots that need to be added in state S in order for every extended room to be in an allowable configuration. We can calculate this quantity recursively as described in the following algorithm.

Algorithm 6 Connect, given a state S find the best solution from S

```
1: best ← ∞
2: for all configurations c such that c is consistent with S and the board do
3:    S' ← new state obtained from S and c
4:    a ← number of new dots obtained by adding c
5:    b ← best solution starting from the state S' {recursive call}
6:    if best > a + b then
7:       best ← a + b
8:    end if
9: end for
10: return best
```

We can implement this general algorithm either by pure dynamic programming or by a recursion with memoization. If the board is of the size (R, C) then there are $N = R/2$ rooms in each column and $M = C/2$ rooms in each row. The total number of states is $NM2^{N+1}$, and so the time complexity of the algorithm is $O(NM2^{N+1})$. In order to reconstruct the solution we will need to keep track of the best configuration in each state, which requires additional $O(NM2^{N+1})$ space.

5 Link

Clearly, we can assume that links are only added to the home page i.e. every new link is placed on the home page and points to some other page. The pages and links form a directed graph with the property that every node has exactly one outgoing link. If we add a link from the home page to a node x, we will say that x is *marked*. If a node can be reached from a marked node in $K - 1$ steps (which means it can be reached from the home page in K steps), we will say that it is *covered*. In this problem we need to find the smallest number of nodes that need to be marked in order to make every node covered.

Consider one connected component of the graph (ignoring the directions of the links). Obviously, there has to be one cycle in the component – every node contains one forward link and by walking along these forward links we have to eventually revisit a node. Also, this cycle is unique, otherwise there would have to be a node with two forward links. Therefore each connected component can be represented as a number of trees (maybe of size one), whose roots are additionally connected to form a cycle.

Pick one component and assume for simplicity that the home page is not inside that component. Let C be the set of nodes of the component lying on the cycle and let T be the set of all other nodes. The algorithm works in two phases. In the first phase, all nodes from T are covered by marking a minimal number of nodes. In the second phase all nodes from the cycle C are covered by marking a minimal number of nodes in C.

We will say that a node x is *significant* if it is not covered yet and cannot be covered by marking an uncovered node other than x. Since nodes from T form a tree, if T is not completely covered there has to be at least one significant node in T. Therefore, a unique minimal covering of T is achieved by successively choosing and marking any significant node in T until all of T is covered.

This approach can be implemented using depth-first search, in which the recursive function finds and marks significant nodes and returns the maximal number of links we can follow downwards before reaching an uncovered node again. It is easy to see that the suggested implementation works in time $O(|T|)$. Again, we are assuming for simplicity that the home page is not the connected component we are considering, that special case can be handled directly.

Algorithm 7 Link, non-cycle nodes, depth-first search

Require: x a node in T, home page not in T
1: $dist \leftarrow 0$
2: **for all** $y \in T$ such that the link from y points to x **do**
3: $dist \leftarrow \max(dist, \text{dfs(y)})$
4: **end for**
5: **if** $dist = 0$ **then** {node x is significant}
6: mark node x
7: return $K - 1$
8: **else** {node x is already covered}
9: return $dist - 1$
10: **end if**

Now we turn our attention to the cycle C. While covering T we may have also covered some nodes in C, now our goal is to cover all the remaining nodes. In general, it is possible that C contains no significant nodes and the same approach as for T therefore does not work. However, as soon as we mark a single node in C, there has to be a significant node (unless we have covered all of C). More precisely, if we mark x in C and walk $K - 1$ steps forward, the next uncovered node has to be significant.

Therefore, if we choose a starting node x, we obtain a candidate minimal covering by walking forward along the circle and marking uncovered nodes as we traverse them. If the length of the cycle is L, there are L choices for x and this algorithm works in time $O(L^2)$. We can speed up this algorithm by noticing that we only need to consider any K successive nodes as a starting point. With this optimization we obtain an $O(KL)$ algorithm for a cycle and a $O(NK)$ algorithm for the entire problem.

Algorithm 8 Link, naive algorithm for a cycle

Require: C a sequence of nodes forming a cycle of length L
1: $best \leftarrow L$
2: **for** $i = 1, \ldots, K$ **do**
3: $curr \leftarrow 0$, $j \leftarrow i$
4: **while** there are uncovered nodes **do**
5: find first uncovered node x starting from j
6: $curr \leftarrow curr + 1$ {mark node x}
7: walk forward $K - 1$ steps, j is the index of the new node
8: **end while**
9: $best \leftarrow \max(best, curr)$
10: **end for**

The solution for the cycle can be further refined by speeding up the traversal

of the cycle. In the naive solution above, after we mark a node we simply walk forward $K-1$ steps and then keep walking until we find an uncovered node. Instead, we can preprocess the cycle and for each node x find what would be the next significant node if x was marked.

Algorithm 9 Link, cycle preprocessing

Require: C a sequence of nodes forming a cycle of length L

```
j ← k
for i = 1, ..., L do
    while c_j is already covered or it can be covered from c_i do
        j ← j + 1 {addition wraps around}
    end while
    Jump[i] ← j
end for
```

In each step of the traversal we jump ahead at least K nodes, and therefore each traversal takes time $O(L/K)$. Since we traverse the cycle K times, we get a total complexity of $O(L)$ for one cycle of length L. Hence, this solution works in total time $O(N)$.

6 Meandian

Let us first figure out what salaries can never be determined. Consider the two lowest paid employees and suppose that we swap their salaries. It is not hard to see that the answer to any possible question would remain the same. Therefore, there is no way to differentiate between the lowest and second lowest paid employee (even if we knew the numbers, we couldn't tell who is who since the salaries are different). Similarly, it is impossible to determine salaries for the two highest paid employees.

Now, let us construct an algorithm that determines the salaries of all other employees. First, consider the case when $N = 5$, we will show how to find the median of the five salaries as well as the index of the corresponding employee. Suppose $a_1, a_2, \ldots, a_5$ are different indices such that $S[a_1] < S[a_2] < \cdots < S[a_5]$ and let us ask all possible questions.

$$\begin{aligned}
b_1 &= \texttt{Meandian}(a_1, a_2, a_3, a_4) = (S[a_2] + S[a_3])/2 \\
b_2 &= \texttt{Meandian}(a_1, a_2, a_3, a_5) = (S[a_2] + S[a_3])/2 \\
b_3 &= \texttt{Meandian}(a_1, a_2, a_4, a_5) = (S[a_2] + S[a_4])/2 \\
b_4 &= \texttt{Meandian}(a_1, a_3, a_4, a_5) = (S[a_3] + S[a_4])/2 \\
b_5 &= \texttt{Meandian}(a_2, a_3, a_4, a_5) = (S[a_3] + S[a_4])/2
\end{aligned}$$

Since all salaries are different, we know that $b_1 = b_2 < b_3 < b_4 = b_5$. Therefore, if we start with any five employees, when we ask all five possible questions and sort the results, we will obtain the sequence $(b)_i$ with this property. The median salary, $S[a_3]$, can now be easily calculated from b_1, b_3 and b_5. The corresponding index, a_3, is simply the index not included in the query that produced b_3 as a result.

Algorithm 10 Meandian, solution for the case when $N = 5$

1: $(b)_i \leftarrow$ answers to all five possible queries
2: sort the sequence $(b)_i$
3: $value \leftarrow b_1 + b_5 - b_3$
4: $index \leftarrow$index not included in the query that produced b_3
5: Salary$[index] \leftarrow value$

To solve the general case we can successively choose any five employees with unknown salaries and run the previous algorithm to find the salary for one of them. After $N - 4$ steps we will have found all possible salaries that can be determined.

Algorithm 11 Meandian, solution for the general case

1: **repeat**
2: choose any five employees with unknown salaries
3: run the algorithm for five employees
4: **until** there are exactly four employees with unknown salaries

In each step we will ask five queries which gives a total of $5(N - 4)$ queries, for $N = 100$ this is well within the given limit of 1000 queries. Note that there are many other ways to correctly solve this problem within the given limit.