7th Asia-Pacific Informatics Olympiad

Hosted by
National University of Singapore, Singapore

Saturday, 11 May, 2013

Task Solutions

Task name	**ROBOTS**	**TOLL**	**TASKSAUTHOR**
Time Limit	1.5s	2.5s	Not Applicable
Heap Size	128MB	128MB	Not Applicable
Stack Size	32MB	32MB	Not Applicable
Points	100	100	100
Number of subtasks	4	5	8
Remark	Submit a program	Submit a program	Submit a data file for each subtask

Scientific Committee

Chang Ee-Chien (Chair)

Chin Zhan Xiong

Steven Halim

Martin Henz

Raymond Kang Seng Ing

Ooi Wei Tsang

Frank Stephan

Sung Wing Kin, Ken

Trinh Tuan Phuong

Harta Wijaya

Task 1 ROBOTS

Task 1: ROBOTS

The engineers at VRI (Voltron Robotics Institute) have built a swarm of n robots. Any two compatible robots that stands on the same grid can merge to form another composite robot.

We label the robots with number 1 to n ($n \leq 9$). Two robots are compatible if they have labels that are consecutive. Originally, each of the n robot has one unique label. A composite robot that is formed after merging two or more robots is assigned two labels, consisting of the minimum and maximum label of the robots that merge into the composite robot.

For example, robot 2 can only merge with robot 3 or robot 1. If robot 2 merges with robot 3, a composite robot 2-3 is formed. If robot 2-3 merges with robot 4-6, a composite robot 2-6 is formed. The robot 1-n is formed when all robots have merged.

The engineers place n robots in a room consisting of $w \times h$ grids, surrounded by walls. Some grids are occluded and cannot be accessed by the robots. Each grid can hold one or more robots, and a robot always occupies exactly one grid. Initially, each robot is placed on a different grid.

The robots are rather primitives. They can move only in a straight line along either the x-axis or y-axis after being pushed by an engineer. After it is pushed in one of the four directions parallel to the x- and y-axis, the robot continues moving, in the direction it is pushed in, until it is blocked either by an occlusion or a wall. After the robot stops moving, it scans for other compatible robots occupying the same grid, and merge with any compatible robot it finds into a larger robot. The merging process continues until no further merging is possible.

To help the robots change direction, the engineers have placed rotating plates in some of the grids. The rotating plates can either rotate in a clockwise or anti-clockwise direction. A robot that moves into a grid with the rotating plate always changes its moving direction by 90 degree in the same direction as the rotating plate. If a robot is being pushed while resting on top of a rotating plate, it rotates by 90 degree before moving off in a straight line, in a direction perpendicular to the direction it is pushed in.

Only one robot can move at one time.

You task is to find the minimum number of pushes such that all n robots are merged together (if possible).

Input

Your program must read from the standard input. The first line of the input file contains three integers, n, w, and h, separated by a space.

The next h lines of the input file describe the room, each line contains w characters. Each of these $w \times h$ characters represents a grid in the room.

A numeric character ('1' to '9') indicates that there is a robot labeled with the corresponding number in the grid. A character 'x' indicates that there is an occlusion in the grid. A character 'A' or 'C' indicates that there is a rotating plate in the grid. 'A' indicates that the plate is rotating anti-clockwise. 'C' indicates that the plate is rotating clockwise. The '.' characters fill all other grid locations.

Output

Your program must write to the standard output, either a single number indicating the minimum number of pushes needed to merge all n robots, or -1 if merging is not possible.

Subtasks

Your program will be tested on four sets of input instances as follow:

1. (10 points) The instances satisfy $n = 2$, $w \leq 10$, $h \leq 10$, with no rotating plates.
2. (20 points) The instances satisfy $n = 2$, $w \leq 10$, $h \leq 10$.
3. (30 points) The instances satisfy $n \leq 9$, $w \leq 300$, $h \leq 300$.
4. (40 points) The instances satisfy $n \leq 9$, $w \leq 500$, $h \leq 500$.

Sample Input

```
4 10 5
1.........
AA...x4...
..A..x....
2....x....
..C.3.A...
```

Sample Output

```
5
```

Description of the sample input/output

The following 5 steps optimally merge the robot.

1. Push robot 3 rightward. The robot moves right, meets a rotating plate, turns anti-clockwise, and continues its movement up. The robot eventually stops in front of the wall.
2. Push robot 4 upward. The robot moves up, stops in front of the wall, and merges with robot 3 to form robot 3-4.
3. Push robot 2 upward. The robot moves up, meets a rotating plate, turns anti-clockwise, hits a wall and stops.
4. Push robot 2 rightward. The robot rotates anti-clockwise, moves up, stops at the corner, and merges with robot 1 to form robot 1-2.
5. Push robot 3-4 leftward. The robot moves left, stops at the corner, and merges with robot 1-2.

APIO 2013: Solution to ROBOTS

Wei Tsang Ooi
National University of Singapore

May 20, 2013

This document explains the solution to the task ROBOTS at APIO 2013. We will present different versions of the solution, starting with simpler but slower version, and progressively improve the running time.

$O(V^2)$ Solution

Here is a quadratic solution that uses dynamic programming in a naive way.

We first pre-compute the minimum number of pushes needed to move any robot from one grid to another. We can solve this as an all-pair shortest path problem on a directed, unweighted, graph $G = (V, E)$, where every vertex is a grid and an edge (u, v) exists if a single push would move the robot from u to v. Denote $c(u, v)$ as the minimum number of pushes to send a robot from u to v. $c(u, v) = \infty$ if it is not possible to reach v from u. This step can be computed in $O(VE)$ time. In the graph we constructed, each vertex has at most four outgoing edges (one for each direction). Therefore, $E = O(V)$ and this step takes $O(V^2)$ time.

Note that we only need to keep track of grid locations that are reacheable by the robots. In the worst case, V is $O(W \times H)$ but in practice, if the number of occlusions are small, $V \ll W \times H$.

We then compute the following: Given two compatible robots at grids u and v, how many pushes are needed if they are to merge in location w? The number of pushes is the sum of the minimum cost from u to w and v to w. The order we push the robots does not matter.

Next, we need to determine the order of merging. The order can be modelled as a binary tree with the individual robots as leaf nodes and composite robots as intermediate nodes. The solution is thus to find, among all possible binary trees of N nodes, the tree that requires the fewest number of pushes. The solution can be achieved with dynamic programming.

Let $d(i, j, u)$ be the minimum number of pushes it takes to merge robots i to j, with the final merge taking place at grid u. The final answer to the task ROBOTS is $\min_{u \in V} d(1, N, u)$.

Let $p(i, j, u)$ be the minimum number of pushes it takes to merge robots i to j in some grid AND move it to grid u. In other words,

$$p(i, j, u) = \min_{v \in V} d(i, j, v) + c(v, u).$$

Here is the dynamic programming solution. First, we consider the base case, and initialize $p(i, i, u)$ to $c(v, u)$ if robot i is located at grid v initiallly.

For the recursive case, we have

$$d(i, j, u) = \min_{i \le k \le j;} p(i, k, u) + p(k + 1, j, u),$$

for $j > i$.

Translating the equations above directly, we have our first solution, presented in Algorithm 1. The algorithm runs in $O(V^2)$ time, with Line 19 and computation of $c(\cdot, \cdot)$ being the bottleneck. Note that since $N \le 9$, we treat the factor N in the running time as a constant.

Algorithm 1 $O(V^2)$ Solution.

```
1: for all i ∈ {1...N} do
2:     for all u ∈ V do
3:         for all j ∈ {1...N} do
4:             d(i, j, u) ← ∞
5:             p(i, j, u) ← ∞
6:         end for
7:         v ← initial position of i
8:         d(i, i, u) ← c(v, u)
9:         p(i, i, u) ← c(v, u)
10:    end for
11: end for
12: for all len ∈ {2...N} do
13:    for all i ∈ {1...N − len + 1} do
14:        j ← i + len − 1
15:        for all u ∈ V do
16:            d(i, j, u) ← min_{k∈i+1...i+N−2} p(i, k, u) + p(k + 1, j, u)
17:        end for
18:        for all u ∈ V do
19:            p(i, j, u) ← min_{v∈V} d(i, j, v) + c(v, u)
20:        end for
21:    end for
22: end for
23: return min_{u∈V} d(1, N, u)
```

$O(VlogV)$ Solution

To improve the running time, we need to find a faster way to compute $p(\cdot, \cdot, \cdot)$ and avoid computing $c(\cdot, \cdot)$. The key insight is that if $p(i, j, v)$ has not been determined, and v is reacheable from some grid location u in one push (i.e., $(u, v) \in E$), then $p(i, j, v) = \min_{u|(u,v)\in E} p(i, j, u) + 1$. Further, we know that if i and j merge at u, then $p(i, j, u) = d(i, j, u)$.

For ease of exposition, we abuse the notation of $p(i, j, v)$ to represent "minimum-so-far" in the following explanation and pseudocode, instead of the minimum.

The improved algorithm to compute $p(\cdot, \cdot, \cdot)$ is similar to that of the Dijkstra's algorithm. We say that we relax a vertex v if we set $p(i, j, v)$ to $p(i, j, u) + 1$ when $p(i, j, v) > p(i, j, u) + 1$ and $(u, v) \in E$. We maintain a priority queue containing the "front" of vertices we are exploring, initialized with vertices corresponding to the merged locations. The main loop expand the front and keep relaxing vertices until there is no more vertices to relax. When this happens, we know that each of the $p(i, j, v)$ already contains its minimum value.

Note that this algorithm to compute $p(i, j, v)$ no longer makes use of $c(u, v)$. It turns out that we do not require $c(u, v)$ during initialization as well, as $p(i, i, u)$ can be initialized by repeatedly relaxing the vertices. The removal of $c(u, v)$ removes the bottleneck $O(V^2)$ from the algorithm.

The pseudocode for this improved algorithm is shown in Algorithm 2. Note that the looping condition at Line 11 has changed to accomodate the calculation of $p(i, i, u)$.

In the worst case, each vertex is inserted into the queue once for each incoming neighbor. Thus, there are $O(E)$ insertions, each taking $O(logV)$ time using a binary heap implementation of a priority queue. Since $E = O(V)$, the running time for this version of the algorithm takes $O(VlogV)$ time.

$O(V)$ Solution

We now present the final solution, relying on the observations that: (O1) the value of $p(\cdot, \cdot, \cdot)$ is bounded and is an integer, and (O2) we always insert an item with a key larger than the current minimum in Line 29.

Let's begin by bounding the value of $p(\cdot, \cdot, \cdot)$. We will present a very loose bound here for simplicity. For any two robots i and j to merge, in the worst case, they must (collectively) visit every reacheable grid (each requiring one push). After merging, the composite robot may be pushed to each of the reacheable grid again, before reaching u. This simple analysis gives an upper bound for $p(i, j, u)$ for any i,j, and u, as $2V$.

Since the key value of the priority queue is an integer and is bounded, we can implement the priority queue as (i) an array of lists $L_1, L_2, ...$, where L_x stores a list of grid locations u such that $p(i, j, u) = x$, and (ii) a value min, such that L_{min} is the non-empty list with the smallest index.

The insert(k,v) operation (Lines 20 and 29) simply appends an item v to the end of the list L_k and updates min if necessary. This operation takes $O(1)$ time. The getMin() operation (Line 25) simply returns an item from the list L_{min} and scans for the next higher min if L_{min} becomes empty. Observation O2 implies that it suffices to make a single pass through all the list (i.e., min always increases inside the while loop starting at Line 24). Summing up the cost of every getMin() operation, the running time is $O(V)$. Therefore, with this implementation of the priority queue, the total running time for the algorithm is reduced to $O(V)$.

Algorithm 2 Priority Queue Solution.

```
1: for all i ∈ {1...N} do
2:     for all u ∈ V do
3:         for all j ∈ {1...N} do
4:             d(i, j, u) ← ∞
5:             p(i, j, u) ← ∞
6:         end for
7:     end for
8:     v ← initial grid position of i
9:     d(i, i, v) ← 0
10: end for
11: for all len ∈ 1...N do
12:     for all i ∈ 1...N − len + 1 do
13:         j ← i + len − 1
14:         Q ← new priority queue
15:         for all u ∈ V do
16:             if len ≥ 2 then
17:                 d(i, j, u) ← min_{k∈i+1...i+N−2} p(i, k, u) + p(k + 1, j, u)
18:             end if
19:             if d(i, j, u) ≠ ∞ then
20:                 Q.insert(u, d(i, j, u))
21:                 p(i, j, u) ← d(i, j, u)
22:             end if
23:         end for
24:         while Q is not empty do
25:             u ← Q.getMin()
26:             for all neighbor v of u do
27:                 if p(i, j, v) > p(i, j, u) + 1 then
28:                     p(i, j, v) = p(i, j, u) + 1
29:                     Q.insert(v, p(i, j, v))
30:                 end if
31:             end for
32:         end while
33:     end for
34: end for
35: return min_{u∈V} d(1, N, u)
```

Credits

Task statement and $O(V^2)$ solution by Wei Tsang Ooi. $O(VlogV)$ solution by Harta Wijaya. $O(V)$ solution by Raymond Kang and Harta Wijaya.

Task 2 TOLL

Task 2: TOLL

Happyland can be described by a set of N towns (numbered 1 to N) initially connected by M bidirectional roads (numbered 1 to M). Town 1 is the central town. It is guaranteed that one can travel from town 1 to any other town through these roads. The roads are toll roads. A user of the road i has to pay a toll fee of c_i cents to the owner of the road. It is known that all of these c_i's are distinct. Recently, K additional new roads are completed and they are owned by a billionaire Mr Greedy. Mr Greedy can decide the toll fees (not necessarily distinct) of the new roads, and he has to announce the toll fees tomorrow.

Two weeks later, there will be a massive carnival in Happyland! Large number of participants will travel to the central town and parade along the roads. A total of p_j participants will leave from town j and travel toward the central town. They will only travel on a set of selected roads, and the selected roads will be announced a day before the event. By an old tradition, the roads are to be selected by the richest person in Happyland, who is Mr Greedy. Constrained by the same tradition, Mr Greedy must select a set of roads that *minimizes the sum of toll fees in the selected set* and yet at the same time allow anyone to travel from town j to town 1 (hence, the selected roads form a "minimum spanning tree" where the toll fees are the weights of the corresponding edges). If there are multiple such sets of roads, Mr Greedy can select any set as long as the sum is minimum.

Mr Greedy is well-aware that the revenue he received from the K new roads does not solely depends on the toll fees. The revenue from a road is actually the total fee collected from people who travel along the road. More precisely, if p people travel along road i, the revenue from the road i is the product $c_i p$. Note that Mr Greedy can only collect fees from the new roads since he does not own any of the old roads.

Mr Greedy has a sneaky plan. He plans to maximize his revenue during the carnival by manipulating the toll fees and the roads selection. He wants to assign the toll fees to the new roads (which are to be announced tomorrow), and select the roads for the carnival (which are to be announced a day before the carnival), in such a way that maximizes his revenue from the K new roads. Note that Mr Greedy still has to follow the tradition of selecting a set of roads that minimizes the sum of toll fees.

You are a reporter and want to expose his plan. To do so, you have to first write a program to determine how much revenue Mr Greedy can make with his sneaky plan.

Input

Your program must read from the standard input. The first line contains three space-separated integers N, M and K. The next M lines describe the initial M roads. The ith of these lines contains space-separated integers a_i, b_i and c_i, indicating that there is a bidirectional road between towns a_i and b_i with toll fee c_i. The next K lines describe the newly built K additional roads. The ith of these lines contains space-separated integers x_i and y_i, indicating that there is a new road connecting towns x_i and y_i. The last line contains N space-separated integers, the j-th of which is p_j, the number of people from town j traveling to town 1.

The input also satisfies the following constraints.

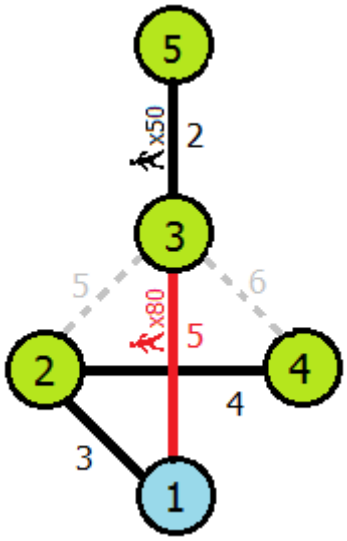
- $1 \leq N \leq 100000$.
- $1 \leq K \leq 20$.
- $1 \leq M \leq 300000$.
- $1 \leq c_i, p_j \leq 10^6$ for each i and j.
- $c_i \neq c_{i'}$, if $i \neq i'$.
- Between any two towns, there is at most one road (including newly built ones).

Output

Your program must write to the standard output a single integer, which is the maximum total revenue obtainable.

Sample Input and Output

Input	Output
5 5 1 3 5 2 1 2 3 2 3 5 2 4 4 4 3 6 1 3 10 20 30 40 50	400



In this sample, Mr Greedy should set the toll fee of the new road (1,3) to be 5 cents. With this toll fee, he can select the roads (3,5), (1,2), (2,4) and (1,3) to minimize sum of toll fees, which is 14 cents. 30 people from town 3 and 50 people from town 5 will pass through the new road to town 1 and hence he can collect an optimal revenue of $(30 + 50) \times 5 = 400$ cents.

If, on the other hand, the toll fee of the new road (1,3) is set to be 10 cents. Now, constrained by the tradition, Mr Greedy must select (3,5), (1,2), (2,4) and (2,3) as this is the only set that minimizes the sum of toll fees. Hence, no revenue will be collected from the new road (1,3) during the carnival.

Subtasks

Your program will be tested on 5 sets of instances as follow:

1. (16 points) $N \leq 10, M \leq 20$ and $K = 1$.
2. (18 points) $N \leq 30, M \leq 50$ and $K \leq 10$.
3. (22 points) $N \leq 1,000, M \leq 5,000$ and $K \leq 10$.
4. (22 points) $N \leq 100,000, M \leq 300,000$ and $K \leq 15$.
5. (22 points) $N \leq 100,000, M \leq 300,000$ and $K \leq 20$.

Solution for Task TOLL

Task author: Raymond Kang Seng Ing
Special thanks: Shen Chuanqi, Assoc Prof Chang Ee-Chien, Dr Sung Wing Kin, Harta Wijaya

Problem

This problem wants us to assign weights to some edges and pick a minimum spanning tree (MST) from the resultant graph to maximize a certain revenue function on the MST. For convenience, we'll call the initially weighted edges "default edges" and the edges we need to price (assign weights to) "Mr G's edges".

Too many MSTs

Given that the number of possible MSTs can be exponential in general graphs, the selection of the MST itself would normally already be intractable. However, in this case it turns out that when the edge weights are *largely* distinct, the number of MSTs is actually not a lot. In fact, notice that if we *fix* some edges to be in the MST, and guarantee that the remaining edge weights are all distinct, then there is only one unique MST. This leads directly to the following observation.

Observation 1. *Suppose we fix some of Mr G's edges to be in the MST, and remove the rest from the graph. Then there's only one possible MST.*

Proof. Contract the connected components formed from Mr G's fixed edges to single vertices. All the edge weights in this graph are now distinct, and the only MST of this graph can be extended to the MST of the original graph by adding in Mr G's edges. □

With this observation, it turns out that once we price those fixed Mr G's edges such that they are included in the MST, there will only be one MST to consider, regardless of the weights of Mr G's edges. This implies that, no matter how we price Mr G's edges, there are only at most 2^K possible MSTs: one for each subset of Mr G's edges to be fixed inside.

How to fix?

Now our problem is: How do we price some edges such that they are included in the MST? One trivial way is simply to price all of them at 0. But our revenue will then also be 0. Indeed, we want to price them as high as possible, while keeping them in the MST. Let us first figure out the maximum weight we can price a single edge, e, to keep it in the MST of any graph.

Observation 2. *If an edge e is the only edge that has the maximum weight in any cycle of a graph, then e cannot be in the MST.*

Proof. Suppose the MST contains e. Remove e from the MST, and we get two disjoint trees that can be connected by an edge e' from that cycle. The weight of e' is strictly less than that of e, thus we can replace e by e' to obtain an MST of smaller total weight, a contradiction. □

Thus, with slight modification, we can come up with a necessary condition for any edge belonging to Mr G to be in the MST:

Condition 1. *The edge must not have the maximum weight in any cycle of the graph, unless it shares the weight with a default edge in the cycle.*

Let us price Mr G's edges so that they will all fulfill Condition 1. Now observe that once they do so, they *must* be in the MST!

Observation 3. *As long as Mr G's edges all fulfill Condition 1, they must be in the MST.*

Proof. Continually find any cycle in the graph and remove the edge of maximum weight since it cannot be in the MST (breaking ties by preferring to remove default edges). Since Mr G's edges all fulfill Condition 1, for each cycle they are contained in they will never be removed. Eventually there will be no cycles and the graph will form an MST with all of Mr G's edges. □

Fulfilling Condition 1

Let us first start with the graph with none of Mr G's edges, and find its MST using a standard MST algorithm like Kruskal's. We shall add in Mr G's edges one by one, ensuring that the edges fulfill Condition 1 all the time. For the first edge e_1, consider the cycle formed from adding e_1 to the initial MST (let's call it the *MST-cycle* for e_1). e_1 cannot be the only edge with maximum weight in this cycle, so we set its weight to be the same as the maximum edge weight in the current MST-cycle, w_1.

Now, what about the other cycles containing e_1? It turns out that there is no need to consider them at all!

Observation 4. *The maximum edge weight in any cycle containing e_1 is no less than w_1, the maximum edge weight in the MST-cycle for e_1.*

Proof. Suppose the maximum edge weight in a cycle containing e_1 is $w < w_1$. Then w_1 is greater than all edge weights in both cycles. It is easy to see that even without Mr G's edge e_1, there must be a new cycle containing w_1 with edges only from these two cycles. But w_1 is the only edge with maximum weight in this new cycle. Hence it cannot be in the initial MST, a contradiction. □

It follows immediately that by pricing e_1 at w_1, it cannot be the only edge with maximum weight in any cycle, fulfilling Conditon 1.

We now price e_1 at w_1 to displace the default edge with weight w_1 from the MST, forming a resultant spanning tree.

Claim. *The resultant tree T is an MST for the new graph that includes e_1.*

Proof. Consider the MST for the graph with only default edges. Every default edge not in this MST must have the maximum edge weight in some cycle and hence cannot be in the MST for the new graph. Since Mr G's edge e_1 fulfills Condition 1, it must be in the MST. Thus the edges in T are the only ones that can be in the MST. □

Now that we have a new "initial MST", we can add in the next edge e_2 similarly. However, now the MST-cycle for e_2 might contain e_1, which we also want to keep inside the MST. If e_1 has the maximum weight in this cycle, e_2 will definitely displace it and hence we must lower the weight of e_1. Naturally, e_1 should be set to at most the next-maximum weight − any larger will mean either e_1 or e_2 has to go! e_2 can then also be set to that same weight, to displace the default edge with that weight.

By lowering the weight of e_1, e_1 will still fulfill Condition 1 for all other cycles. The argument that the other cycles containing e_2 do not need to be considered is similar to that for e_1; the only difference is that the initial MST now contains e_1. Thus e_2 also fulfills Condition 1. The resultant tree is now an MST for the graph with e_1 and e_2 added in, by similar argument.

Hence, in a similar vein, we can go on to price all of Mr G's edges maximally. Eventually, they will all fulfill Condition 1 − and along the way, we have also built up an MST containing all of them!

We now have a simple solution for our original task: Iterate through all subsets of Mr G's edges to fix in the MST by adding in the edges one by one in a depth-first manner. Each time an edge is added in, find its MST-cycle and set its weight to be the maximum default edge weight w_{max} in the cycle. Mr G's edges in the cycle cannot have weights greater than w_{max}, so update all their weights too. After adding in the edges for each subset, we have an MST whose revenue can be computed using a simple tree traversal. The runtime is $O(M \log M + 2^K N)$. This should solve subtask 3 to get 56 points.

Compressing the tree

Our initial MST is huge: a total of $N - 1$ edges, with N up to $100,000$. However, it turns out that not many of these default edges are important. If we look at our simple solution, the only default edges we need to consider are those that are maximum in some MST-cycle when we add in Mr G's edges. We shall call these "maximal default edges". It turns out that there are only at most K of them!

Observation 5. *The maximal default edges for a given subset of Mr G's edges must be part of the maximal default edges for the full set of Mr G's edges.*

Proof. We can add in Mr G's edges in arbitrary order to obtain the same set of maximal default edges. Thus, when adding in the full set of Mr G's edges, we add in the given subset of Mr G's edges first to obtain the maximal default edges for that subset. □

So how do we find the maximal default edges for the full set of Mr G's edges? We know that they cannot be in the MST with all K Mr G's edges, and yet are present in the initial MST with only default edges. Hence, we can easily find them by computing both MSTs and doing a quick comparison between them. This takes $O(M \log M)$ time.

Once the maximal default edges are obtained, the initital MST can now be compressed to at most K edges connecting $K + 1$ components, using a simple union-find data structure. Our simple solution now finds the MST-cycle, prices the weights, and computes the revenue in $O(K)$ time. Hence the runtime is $O(M \log M + 2^K K)$. This is expected to obtain 100 points.

Additional remarks

If we do not add in Mr K's edges one by one in a depth-first manner, but instead recompute the pricing for all edges for each subset of Mr K's edges, an additional $O(K)$ time is invoked. This gives solutions of $O(M \log M + 2^K NK)$ and $O(M \log M + 2^K K^2)$ depending on whether the tree compression is done. The former should still solve subtask 3, while the latter only solves subtask 4 to get 78 points.

There is another solution for subtasks 3 and 4 that does not involve adding in Mr G's edges one by one: For each subset of Mr G's edges, compute the MST with the entire subset fixed inside from scratch. Now use all K edges that are present in the initial MST but not present in this MST to update the edge weights of Mr G's edges to fulfill Condition 1. The runtime is $O(M \log M + 2^K NK)$ or $O(M \log M + 2^K K^2)$ depending on whether the compression was done. This approach is more widely used, but more difficult to speed up to the model runtime to get full score. Nevertheless, it is possible to do so using specific data structures to quicken the updating of the edge weights. The only contestant to solve subtask 4 used this method.

Credits

Initial task statement by Raymond Kang Seng Ing.
Refinement of task statement by Assoc Prof Chang Ee-Chien.
Solution sketch by Raymond Kang Seng Ing and Shen Chuanqi.
Verification of proofs by Assoc Prof Chang Ee-Chien and Dr Sung Wing Kin.
Additional testing and verification by Harta Wijaya.

Task 3 TASKSAUTHOR

Task 3: TASKSAUTHOR

There are many programming contests in the world today. Setting a good programming contest task is not easy. One challenge is the setting up of *test data*. A good test data should be able to differentiate a code that meets the goals, from another seemingly correct code that fails in some special cases.

In this task, your role in a contest is reversed! As an experienced programmer, you are helping the Happy Programmer Contest's committee in setting up their test data. The committee has selected two graph problems with a total of 8 different subtasks, and has written a few codes that seemingly solve the graph problems. In designing a subtask, the committee has the intention that some of the codes would get all the points, whereas some would gain zero or some points. You are given all those codes in C, C++ and Pascal versions[1]. For each subtask, your job is to produce a test data X that *differentiates* two given codes, code **A** and code **B**. More specifically, the following two conditions must be met:

1. On input X, code **A** must not lead to Time Limit Exceeded (TLE).
2. On input X, code **B** must lead to TLE.

In addition, the committee prefers smaller test data, with a target of at most T integers in the test data.

The two problems selected by the committee are the Single-Source Shortest Paths (SSSP) problem, and a graph problem we called the Mystery problem. The pseudo-codes of the codes written by the committee are listed in the appendix and the C, C++ and Pascal implementations can be found in the attached zip file that accompanies task 3 in the grading server.

Subtasks

Please refer to Table 1. Each row describes a subtask. Note that subtask 1 to 6 are on the SSSP problem whereas subtask 7 and 8 are on the Mystery problem. The number of points allocated for the subtasks are listed in column S.

Subtask	Points S	Target T	Problem	Code **A**	Code **B**
1	3	107	SSSP	ModifiedDijkstra	FloydWarshall
2	7	2222	SSSP	FloydWarshall	OptimizedBellmanFord
3	8	105	SSSP	OptimizedBellmanFord	FloydWarshall
4	17	157	SSSP	FloydWarshall	ModifiedDijkstra
5	10	1016	SSSP	ModifiedDijkstra	OptimizedBellmanFord
6	19	143	SSSP	OptimizedBellmanFord	ModifiedDijkstra
7	11	3004	Mystery	Gamble1	RecursiveBacktracking
8	25	3004	Mystery	RecursiveBacktracking	Gamble2

Table 1: The 8 Subtasks.

[1] All codes implement algorithms that are permitted inside the IOI syllabus.

To get any point for a subtask, your test data X must be able to differentiate the corresponding code **A** and code **B**. In addition, the number of points you received depends on the number of signed integers in X. Suppose X contains F integers, S points are allocated to the subtask, and T is the targeted size, then the number of points awarded is calculated as follow:

$$\lfloor (S/100) \times \lfloor min(100 \times T/F, 100) \rfloor \rfloor.$$

where $\lfloor\ \rfloor$ denotes the rounding down operation. Hence, if your test data X contains not more than T integers, the full S points are awarded.

Grading

You must name each of your eight test data as `tasksauthor.outX.1` where `X` is the subtask number. Before submission, you are required to compress your test data files using `gzip`. In Unix systems, the following command produces the compressed file:

```
tar -cvzf tasksauthor.tgz tasksauthor.out*.1
```

In Windows systems, use software like 7-Zip or Winzip to produce this tar-gzipped archive. Submit only the file `tasksauthor.tgz` to the grading server.

The grading server will unpack the compressed file. For each subtask, say subtask `X`, the grader carries out the following steps to determine the number of points to be awarded for subtask `X`:

C1. If test data `tasksauthor.outX.1` does not exist, halts and no point will be awarded.

C2. Checks the format of `tasksauthor.outX.1`.
If the input format is invalid, halts and no point will be awarded.

C3. Runs code **A** with `tasksauthor.outX.1` as the input.
If TLE is triggered, halts and no point will be awarded.

C4. Runs code **B** with `tasksauthor.outX.1` as the input.
If TLE is triggered, halts and awards a number of points calculated using the formula:

$$\lfloor (S/100) \times \lfloor min(100 \times T/F, 100) \rfloor \rfloor.$$

All the provided codes maintain a counter (the variable `counter`) that keeps track of the number of performed operations. During the execution of a code, when the value of the counter exceeds 1,000,000, then we consider the code has triggered TLE.

Problem Statement 1: Single-Source Shortest Paths (SSSP)

Given a directed weighted graph G and two vertices s and t in G, let $p(s,t)$ be the shortest path weight from the "source" s to the "destination" t. If t is not reachable from s, then $p(s,t)$ is defined to be 1,000,000,000. In this problem, the input is the graph G and a sequence of Q queries $(s_1,t_1),(s_2,t_2),\ldots,(s_Q,t_Q)$. The output is the corresponding query results $p(s_1,t_1),p(s_2,t_2),\ldots,p(s_Q,t_Q)$.

Input/Output file Format

The input file consists of two blocks. The first block describes the adjacency list of a directed weighted graph G. The second block describes shortest path queries on G.

The first block starts with an integer V in one line, which is the number of vertices in G. The vertices are labelled as $0,1,\ldots,V-1$. Then, V lines follow where each line corresponds to a vertex, starting from vertex 0. Each line starts with n_i that describes how many outgoing-edges the vertex i has. Next, n_i pairs of integers (j,w) follow where each pair corresponds to an outgoing-edge. The first integer j in a pair is the label of the vertex the edge points to, and the second integer w is the edge's weight.

The second block starts with an integer Q in one line. Next, Q lines follows. The k-th line contains two integers s_k and t_k, corresponding to the source and destination vertex respectively.

Any two consecutive integers in one line must be separated by at least one space. Additionally, the input satisfies the following:

1. $0 < V \leq 300$,
2. n_i is a non-negative integer $\forall i \in [0..V-1]$,
3. $0 \leq j < V$,
4. $|w| < 10^6$ where $|w|$ denotes the absolute value of w,
5. $0 \leq \sum_{i=0}^{V-1} n_i \leq 5000$,
6. $0 < Q \leq 10$,
7. $0 \leq s_k < V, 0 \leq t_k < V, \forall k \in [1..Q]$, and
8. the graph G must **not** have any negative weight cycle.

Recall that the grading server will check for the above constraints in step C2.

The output file format is less relevant in this task. Anyway, the output consists of Q lines, and the k-th line contains the integer $p(s_k,t_k)$. For convenience, the provided codes will print out the value of the variable `counter` at the end of the output.

Sample Input File[2]

```
3
2 1 4 2 1
0
1 1 2
2
0 1
1 0
```

Sample Output File[3]

```
3
1000000000
The value of counter is: 5
```

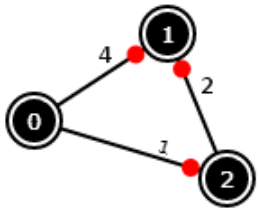


Figure 1: Directed Weighted Graph from the Sample Input File

[2]There are fifteen integers in this input file, therefore $F = 15$.
[3]The value of counter is 5 when the sample input file given above is run on ModifiedDijkstra.cpp/pas.

Problem Statement 2: Mystery

Given an undirected input graph G with V vertices and E edges, label each vertex in G with an integer $\in [0..(X-1)]$ so that no two endpoints of any edge in G has the same label. The value of X must be the lowest possible for graph G.

Input/Output file format

The input file starts with two integers V and E in one line. Then, E lines follows. Each line contains two integers a and b that denotes an *undirected* edge (a, b) in G. In addition, the input satisfies the following constraints (to be checked in step C2):

1. $70 < V < 1000$,
2. $1500 < E < 10^6$, and
3. for any edge (a, b), we have $a \neq b$, $0 \leq a < V$, $0 \leq b < V$, and it appears only once in G.

The output file starts with an integer X in one line, the smallest integer so that vertex labelling is feasible. The next line contains V integers that describe the integer label of vertex 0, vertex 1, ..., vertex $V-1$, and the last line is the value of `counter`.

Sample Input File[4]

```
4 5
0 1
0 2
0 3
1 2
2 3
```

Sample Output File[5]

```
3
0 1 2 1
The value of counter is: 18
```

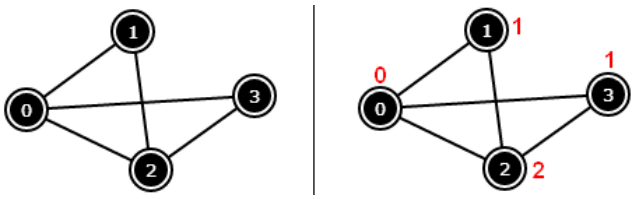


Figure 2: Left: Undirected Graph from the Sample Input File; Right: Its Labels with $X = 3$

[4]There are twelve integers in this input file, therefore $F = 12$. However, this small sample input file is only used for illustration. It is not valid as its V and E values are too small.

[5]The value of counter is 18 when the sample input file given above is run on RecursiveBacktracking.cpp/pas.

Appendix: Pseudo-codes

Here are the algorithms of the provided codes. The variable `counter` "approximates" the runtime by keeping track of some operations. Our grading server uses the C++ version of these implementation codes.

FloydWarshall.cpp/pas

```
// pre-condition: the graph is stored in an adjacency matrix M
counter = 0
for k = 0 to V-1
   for i = 0 to V-1
      for j = 0 to V-1
         increase counter by 1;
         M[i][j] = min(M[i][j], M[i][k] + M[k][j]);
for each query p(s,t)
   output M[s][t];
```

OptimizedBellmanFord.cpp/pas

```
// pre-condition: the graph is stored in an adjacency list L
counter = 0
for each query p(s,t);
   dist[s] = 0; // s is the source vertex
   loop V-1 times
      change = false;
      for each edge (u,v) in L
         increase counter by 1;
         if dist[u] + weight(u,v) < dist[v]
            dist[v] = dist[u] + weight(u,v);
            change = true;
      if change is false // this is the 'optimized' Bellman Ford
         break from the outermost loop;
   output dist[t];
```

ModifiedDijkstra.cpp/pas

```
// pre-condition: the graph is stored in an adjacency list L
counter = 0;
for each query p(s,t)
   dist[s] = 0;
   pq.push(pair(0, s)); // pq is a priority queue
   while pq is not empty
      increase counter by 1;
      (d, u) = the top element of pq;
      remove the top element from pq;
      if (d == dist[u])
         for each edge (u,v) in L
            if (dist[u] + weight(u,v) ) < dist[v]
               dist[v] = dist[u] + weight(u,v);
               insert pair (dist[v], v) into the pq;
   output dist[t];
```

Gamble1.cpp/pas

```
Sets X = V and labels vertex i in [0..V-1] with i;
Sets counter = 0; // will never get TLE
```

Gamble2.cpp/pas

```
Sets X = V and labels vertex i in [0..V-1] with i;
Sets counter = 1000001; // force this to get TLE
```

RecursiveBacktracking.cpp/pas

```
This algorithm tries X from 2 to V one by one
and stops at the first valid X.

For each X, the backtracking routine label vertex 0 with 0,
then for each vertex u that has been assigned a label,
the backtracking routine tries to assign
the smallest possible label up to label X-1 to its neighbor v,
and backtracks if necessary.

// Please check RecursiveBacktracking.cpp/pas to see
// the exact lines where the iteration counter is increased by 1
```

Task 3: `Task Author - Solution Sketch`

Problem author: Dr Steven Halim (Singapore IOI team leader)
Main tester: Harta Wijaya (ACM ICPC World Finalist 2012-2013)
Special thanks: Associate Professor Chang Ee Chien,
Raymond Kang Seng Ing (IOI Gold Medalist 2010-2011),
Koh Zi Chun (IOI Bronze 2005, ACM ICPC World Finalist 2012),
Dr Felix Halim (IOI 2002 and ACM ICPC World Finalist 2007).

Subtasks 1-6 Overview

Subtask 1-6: In these six subtasks, contestants are asked to compare and to contrast three shortest paths algorithms. Contestants need to really understand how these algorithms work in order to get high points in this task. Observe that all three algorithms, the $O((V + E) \log V)$ ModifiedDijkstra[1], the $O(VE)$ OptimizedBellmanFord, and the $O(V^3)$ FloydWarshall can solve the **general** SSSP problem without problem. Thus, their runtime differences should be due to some **special cases**.

We can solve the easiest two Subtasks 1 & 3 with the same test data by concentrating on making FloydWarshall TLE due to its huge $O(V^3)$ time complexity. And similarly we can group Subtasks 2 & 5 by concentrating on making OptimizedBellmanFord TLE. Finally, we group Subtasks 4 & 6 by concentrating on making ModifiedDijkstra TLE. Therefore, there are actually only[2] *three* distinct subtasks instead of *six* for the SSSP problem. Notice that the values of S and T for these related subtasks are made 'slightly different' to act as superficial camouflage so that *some* contestants do not immediately see this fact.

Subtasks 1-6 'Complicated' Constraints Explained

We disallow any negative cycle[3] as it can cause the solution for Subtask 2-5 and 4-6 to be trivial (i.e. give any small negative cycle to make the OptimizedBellmanFord runs in exactly $O(VE)$ and to make the ModifiedDijkstra runs into infinite loop).

However, to check for the presence of negative cycle, we need to use a special checker, which is our own $O(VE)$ OptimizedBellmanFord algorithm. We cannot afford to have contestants submitting large test case. Therefore we need to restrict V and $E = \sum_{i=0}^{V-1} n_i$ of Subtasks 1-6 to maximum 300 and 5000, respectively. This way, the largest $V \times E = 300 \times 5000 = 1500000$ operations, which is fast enough.

The values of j, s_k, t_k must be $\in [0..V-1]$ for a valid test case. The reason is obvious.

We restrict the absolute values of the edge weight w to be less than 10^6 to avoid unnecessary detours to BigInteger operations due to overflow cases. Notice that the largest valid SSSP value

[1] We are using the implementation shown in Steven's 'Competitive Programming' textbook, which is from: http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=standardTemplateLibrary2#dijkstra2

[2] The best solutions for subtask 2 and 5 are slightly different.

[3] The wording in the task statement is slightly different, i.e. negative cycle reachable from a source in the query block. This clarification is raised by only three contestants from Australia, Singapore, and Japan.

if a graph has at most 300 vertices is $(300-1) \times 10^6 = 299000000$ which still fits in the standard 32-bit signed integer (int in C/C++, Longint in Pascal).

Lastly we restrict Q to maximum 10 to avoid a situation in Subtask 4 where the $O(V^3)$ FloydWarshall is used once and then many $O(1)$ queries are asked such that the ModifiedDijkstra gets TLE without using the intended solution below. Q has to be allowed to be > 1 as that is the only way to make the OptimizedBellmanFord gets TLE in Subtask 2 & 5.

Subtasks 1 & 3

The solution for Subtask 1 (and 3) is very easy[4]. As a code is judged as TLE if it requires more than 1000000 (1 million) operations and FloydWarshall runs in $O(V^3)$, we just need to create a graph with $V > 100$ (the smallest one is $V = 101$), zero edge, and one $s - t$ query: '0 0'. This input graph causes FloydWarshall to execute $101^3 = 1030301 > 1000000$ operations $\rightarrow$ TLE and the trivial $s - t$ $(0 - 0)$ query produces a trivial answer $p(s, t) = 0$. The $O((V + E) \log V)$ ModifiedDijkstra for Subtask 1 and $O(VE)$ OptimizedBellmanFord for Subtask 3 will pass such test case without issue. This is an easy 3+8 = 11 points. Contestants who bother to read and understand the task statement should be able to get at least these 11 points. More than half official contestants manage to do this (see Figure 2).

Subtasks 2 & 5

Under the task constraints, the only way to make OptimizedBellmanFord gets TLE is to run multiple queries at its worst case $O(VE)$ per query so that it exceeds 1000000 operations.

The OptimizedBellmanFord has a check inside the outer for-loop that if there is no more edge relaxation, it will immediately stop. In order to force OptimizedBellmanFord to really repeat all E edges relaxation $V - 1$ times, we need to observe the listing of the E edges. The code is written in such a way that all outgoing edges of vertex 0 is processed first, then all outgoing edges of vertex 1, ..., until all outgoing edges of vertex $V - 1$. Therefore, we need the source vertex to be vertex $V - 1$. We create a graph where we have edges from vertex i to vertex j if $j < i$. The weight of edge (i, j) is 1 if $i - j = 1$ or $i - j + 1$ otherwise. This way, all E edges have to be relaxed exactly $V - 1$ times. Setting $Q = 10$ and $V = 60$ is enough as $10 \times (60 - 1) \times 60 \times (60 - 1)/2 = 1044300 > 1000000$ operations $\rightarrow$ TLE. We need to run OptimizedBellmanFord $Q = 10$ times as that is the only way for a $O(QVE)$ algorithm to beat a $O(V^3)$ algorithm (in Subtask 2) because FloydWarshall only process the AdjacencyMatrix in $O(V^3)$ once and can answer subsequent SSSP queries in $O(1)$. The ModifiedDijkstra in Subtask 5 has no problem with this input graph.

The input file above uses is 3622 integers. For Subtask 2, this score $\approx \lfloor min(2222/3622, 1) \times 7 \rfloor = 4$ points. There are several more minor optimizations possible along this line of thinking. Our best answer using this approach contains 2222 integers.

With the 'multigraph' trick (allowing more than one edges between two vertices – which is not restricted), we can produce a smaller input file that uses only 2144 integers with the

[4]The original draft of TASKSAUTHOR that includes checking for 'Wrong Answer' condition allow for more tricky solution, but this is not used in the actual competition.

following code. The key idea is to place unnecessary edges between vertex 0 to vertex 1 with weight 1 such that the total operation count of OptimizedBellmanFord exceeds 1000000. This solution will score the full 7 points for Subtask 2 but will only get $\approx \lfloor min(1016/2144, 1) \times 10 \rfloor = 4$ scores in Subtask 5.

```
#include <cstdio>
#include <vector>
using namespace std;
#define REP(i, n) for (int i = 0, _n = n; i < _n; i++)
#define MAXN 300
vector< pair<int,int> > con[MAXN];

int main() {
  freopen("tasksauthor.out5.1", "w", stdout);
  int V = 100, m = 1011, tot = 0;
  printf("%d\n", V);
  // create a link list from vertex V-1 to vertex 0
  REP (i, V-1) {
    tot++;
    con[i+1].push_back(make_pair(i, 1));
  }
  // dump many unnecessary edges between 0->1 with weight 1
  while (tot < m) {
    tot++;
    con[0].push_back(make_pair(1, 1)); // multigraph
  }
  REP (i, V) {
    printf("%d", (int)con[i].size());
    REP (j, (int)con[i].size())
      printf(" %d %d", con[i][j].first, con[i][j].second);
    printf("\n");
  }
  int Q = 10;
  printf("%d\n", Q);
  REP (i, Q)
    printf("%d 0\n", V-1); // ask from vertex V-1 to vertex 0
  return 0;
}
```

The last 6 points of Subtask 5 requires a small change which will not work for Subtask 2. We can change V from 100 to 282 vertices and set m accordingly. This way, we can get the input file with only 1016 integers which scores the full 10 points for Subtask 5. This should be the first medal differentiator. See Figure 2, the 7+10 = 17 points separate contestants who scored 64 and slightly below 47.

Subtasks 4 & 6

The solution for these two subtasks is quite hard to get and likely found as novel by many contestants. It can be rather surprising for some contestants that the faster ModifiedDijkstra (which is a Dijkstra implementation found in TopCoder can get TLE on a certain input graph whereas the supposedly slower FloydWarshall and BellmanFord can get AC—without the presence of any negative cycle. For Subtask 4, most contestants will think along making the queries as many as possible (limited 10 queries) so that 10 times ModifiedDijkstra is slower than one FloydWarshall. This will not work.

During testing, we ourselves needed around 30-50 minutes to figure out the required input graph structure. Those who are familiar with ModifiedDijkstra should immediately realize that it has to be related with the fact that ModifiedDijkstra re-enqueue and re-process new vertex information pair. On a graph with negative weight but no negative weight cycle, one can construct a 'dual path' graph[5] as shown in Figure 1 so that ModifiedDijkstra reprocess many vertices many times (exponentially), whereas FloydWarshall simply runs in $O(V^3)$ in Subtask 4 and OptimizedBellmanFord simply runs in $O(VE)$ in Subtask 6.

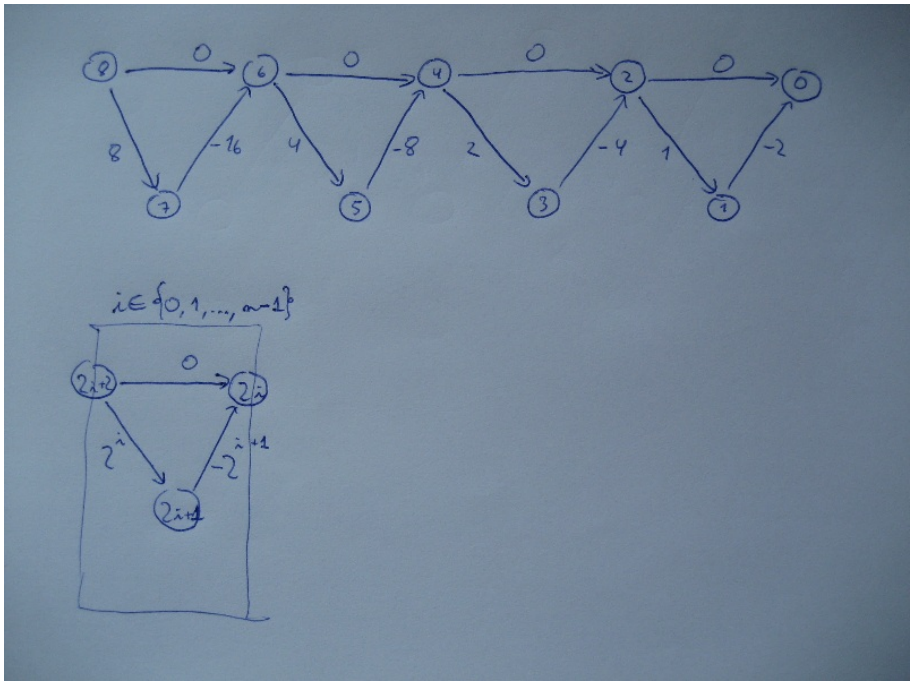


Figure 1: 'Dual path' graph with negative weight edges

Our initial input file as shown in Figure 1 above uses 157 integers which will get the full 17 points for Subtask 4.

However, there are a few optimization tricks that we have included in Subtask 6. The Subtask 4 solution will only get $\approx \lfloor min(143/157, 1) \times 19 \rfloor = 17$ points in Subtask 6. The last 2

[5]This test case is made known to the author by Francisco Criado from Spain via email on 24 Dec 2012.

points require several optimizations which is probably very hard to get under time constrained contest setting.

Solving Subtask 4 & 6 should give a contestant a significant 17+19 = 36 points advantage compared to other contestants who cannot solve them. This is very evident in Figure 2. Many contestants who scored greater than 64 points are the ones who managed to crack these two subtasks. Those who scored beyond 64 points are likely either a silver or a gold medalist in this APIO 2013.

Subtasks 7-8 Overview

Subtask 7 $\&$ 8 is about NP-hard problem: Graph k-Colorability (Chromatic Number)[6]. Some contestants may already have a priori knowledge about this problem that may or may not give them advantage. However, we expect that majority of high school students taking part in APIO 2013 have never heard about this problem before. The name of this problem is changed to 'Mystery' and the term 'graph coloring' have all been changed to 'labeling' to superficially camouflage this fact.

Subtasks 7-8 'Complicated' Constraints Explained

We first specify $70 < V < 10^6$ and $1500 < E < 10^6$ to avoid contestant giving us a very trivial graph, e.g. a line graph with 2 vertices, 1 edge, and $X = 2$, with label 0 and 1.

To make the problem consistent with the standard Chromatic Number problem, we restrict $a! = b$, $0 \leq a < V$, $0 \leq b < V$, and edge (a, b) can only appear at most once to prevent multigraphs and self-loops.

Subtasks 7

The program Gamble1 (which will never get TLE) sets Chromatic Number to be exactly V and then label vertex 0-1-2-3-...-(V-1) with 0-1-2-3-...-(V-1), respectively. A complete graph of size V = 71 definitely have this pattern, but it uses 4972 integers. RecursiveBacktracking will eventually get the same answer, but it will be very slow (TLE and rejected). However, this is not the smallest input graph. A clever generator with $X = V$ as shown in Subtasks 8 below can produce an input file with 3004 integers. Without this better solution, contestant that submits a complete graph solution will only score $\approx \lfloor min(3004/4972, 1) \times 11 \rfloor = 6$ points.

Note that since we do not do any WA versus AC test for Gamble1, actually submitting any graph with $V > 70$ and $E > 1500$ will give contestant some point and submitting a graph with $V = 71$ and $E = 1501$ will give contestant the full 11 points, i.e. this is another giveaway subtask.

[6]The original version of this problem is the NP-complete version of Graph Coloring with fixed X and ask if there exists labeling with at most X colors. We test AC versus WA by polynomially evaluating the answer given by contestants. However, this WA check feature is dropped from the final version to make the task simpler.

Subtasks 8

This subtask can be difficult—but not as difficult as Subtasks 4 and 6. The program Gamble2 is always set to get TLE. Therefore, this subtask becomes a task of finding a valid input graph with $V > 70$ and $E > 1500$ such that RecursiveBacktracking does not get TLE.

The easiest solution is to relate this problem to a classic game: Sudoku. Some of the contestants may have played this Sudoku puzzle game. If they can link this puzzle to the requirements of this problem, this problem will be easy. Build a 9x9 Sudoku Graph. There are $V = 81$ vertices and $E = V * (16 + 4) = 20V = 20 * 81 = 1620$ edges in Sudoku graph which satisfy the required constraints. This graph is highly structured and if vertex numbers are properly placed, RecursiveBacktracking will not have much problem getting the answer. The Sudoku graph uses 3242 integers and will score $\approx \lfloor min(3004/3242, 1) \times 25 \rfloor = 23$ points.

To get the full 25 points, contestant can use the following easy test case generator as it produces an input file that contains 3004 integers.

```
#include <cstdio>
using namespace std;

int n, m, col[1010];

int main() {
  freopen("tasksauthor.out8.1", "w", stdout);
  n = 71; // put these n vertices in a straight line
  m = 1501;
  printf("%d %d\n", n, m);
  for (int i = 0; i < n; i++) // X = 3, three colors
    col[i] = i % 3; // give color 0,1,2,0,1,2,0,1,2,...
  int cnt = 0;
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++)
      if (col[i] != col[j]) { // if the have to be different
        printf("%d %d\n", i, j); // give an edge
        cnt++;
        if (cnt == m) break;
      }
    if (cnt == m) break;
  }
  return 0;
}
```

Getting 11+25 = 36 points (which is the same as the total score of Subtask 4 & 6) will be a huge boost for a contestant. This typically brings contestant from 11 points to around 45-47 points (see Figure 2).

Post Contest Remarks

This is a scientific experiment for APIO community. The style of this task is indeed surprising for many team leaders and contestants. However, looking at the call for tasks of IOI 2013 `http://www.ioi2013.org/competition/call-for-tasks/`, we feel that the OI community around the world have to be expect such surprises in IOI 2013 and beyond. We quote two paragraphs from that call for tasks below:

"We are particularly interested in tasks whose basic rules (if not optimal strategy) are accessible to a wide audience, and tasks that illustrate algorithms and computational problems that arise in a variety of human endeavours. Open-ended tasks — ones that do not necessarily have a known efficient or optimal solution — are welcome.

We are also particularly interested in tasks that go beyond the typical format in which a program collects input, performs some computation, and returns output. Examples include 'reactive' and 'output only' tasks which have been used occasionally in previous IOIs. Tasks with some measure of solution effectiveness other than CPU time consumption are encouraged."

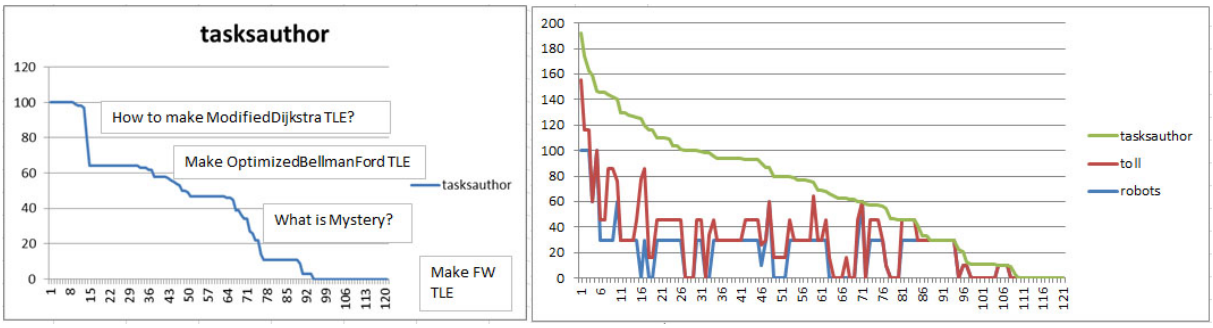


Figure 2: The point distribution for TASKSAUTHOR