Shortest paths contest problems analysis

Gleb Evstropov
evstropovgleb@gmail.com

March 13, 2023

1 BFS

Just implement BFS. Constraints are small enough so you do not have to worry about your solution efficiency at all.

2 Shortest Path

Implement $O(n^2 + m)$ version of Dijkstra's algorithm.

3 Dijkstra

Implement $O(n \log n + m)$ version of Dijkstra's algorithm.

4 Clocks

Transform this problem into a shortest path problem in directed unweighted graph.

Let each state `HH:MM` of the clock be a vertex of the graph. There are $24 \times 60 = 1440$ states. Pressing a button at some state v corresponds to traversing an arc. For example, pressing button (a_1, b_1) at state (h, m) changes the state into $((h+a_1)\%24, (m+b_1)\%60))$. It is similar to traversing the arc from $v(h, m)$ to $v((h + a_1)\%24, (m + b_1)\%60)$.

After we construct the graph representing the clocks' states and button usages, we just compute the shortest path from (h_1, m_1) to (h_2, m_2).

5 Strange Game

The problem statement suggest a pretty trivial encoding. The state of the game equal to s corresponds to vertex s. Possible moves correspond to arcs.

While doing BFS one should also compute $from$ array (see lecture notes) in order to be able to restore the solution afterwards.

6 Dijkstra But Cooler

For this particular problem it will work to assign $w(uv) = w(v)$. Then just run Dijkstra's algorithm.

However, there is a standard way to approach weighted vertices that will help you in many other tasks. Split each vertex v in two vertices v_{in} and v_{out}. Draw an arc from v_{in} to v_{out} of weight $w(v)$. If there was an arc uv in the original graph, it should be transformed into $u_{out}v_{in}$ arc in the new graph. If there was a bidirectional edge uv in the original graph, it will generate two edges $u_{out}v_{in}$ and $v_{out}u_{in}$ in the new graph.

7 Raiffeisenbank Logistics

Let's first consider the inputs with all t_i distinct.

Denote as $d_i(v)$ the minimum number of changes you need to make in order to be able to get from spot 1 to spot v using first i programs only. $d_i(v) = -1$ if this is impossible.

$d_m(n)$ is the answer. How do we compute d_i values if we know d_{i-1}?

1. First set $d_{i-1}(v) = d_i(v)$ for all v.
2. Denote the i-th edge as u_iv_i.
3. Update $d_i(v_i) = \min(d_i(v_i), d_i(u_i))$.
4. Update $d_i(u_i) = \min(d_i(u_i), d_i(v_i) + 1)$.

The update happens in constant time, so the solution takes $O(m)$ after we sort all programs by t_i.

How do we solve the problem in case of equal t_i? It doesn't change much, actually, but you can't just apply all the changes directly to array d now. Instead you need to determine a group of edges with the same value of t_i and process them altogether. That should be done in two phases.

1. Consider all edges and try to update values of d in the same way as before. However, write its result into a new array d'.
2. Copy back to d all $d'(v)$ that were subject to updates.

8 Cycle of negative weight

The problem just asks you to find a cycle of negative weight. Please see the lecture notes for details on how to do this.

9 Transportation

Skipping all the complications with time and weight units the problem boils down to the following. You are given graph G, each edge is assigned two values $t(e)$ and $w(e)$, they are the time it takes to traverse this edge and weight limit, respectively. You need to compute the maximum possible weight x, such that there exists a path from 1 to n that takes no more than T seconds to traverse in total, and all edges of this path have weight limit $\leq X$.

Observation 1. For a fixed value of x we can compute the minimum time required to get from 1 to n with one run of Dijkstra's algorithm. Denote this value as $f(x)$. We need to find maximum x such that $f(x) \leq T$.

Observation 2. $f(x)$ is monotonous, i.e. $f(x) \leq f(x+1)$.

Thus, we can do binary search for a maximum possible value of x such that $f(x) \leq T$.

The overall complexity of this solution is $O(m \log n \log C)$.

10 Random's Games

In this problem we need to run a bidirectional BFS. That means start two BFS's, one from vertex u and one from vertex v, but instead of doing this consecutively, simulate them in parallel step-by-step.

Maintain two queues and two arrays with distances, one for BFS from u and one from BFS from v. Alternate extracting a vertex from the first queue and from the second queue.

There is a tricky part about the stop conditions. You can try three of them.

1. There is a node that is visited by both BFS.
2. There is a node that is processed (extracted from the queue) by both BFS.
3. There is a node that is visited by one BFS and processed by the other.

One of them is wrong. One of the is too slow. Choose smart!

Note that this task is very sensitive to the efficiency of your implementation. Consider using your own array-based implementation of a queue. If it still doesn't fit in time limit, consider some careful node processing after you find the first node seen by both BFS. No need to add new nodes into the queue for example, prune the already existing queues of the nodes that definitely do not need to be processed.

Do not forget to check whether two query nodes belong to the same connected component before starting BFS.

The expected running time is $O(q\sqrt{m})$.

11 Don't Annoy the Mayor

This problem has a lot of corner cases that one should be careful not to miss.

1. If there is an edge of negative weight you can just go strictly to it, traverse it as many times as you want and then go to finish. The answer would be 0.

2. Otherwise, you can try and make some edge negative. Try all of them, calculate how much it would cost to make it negative, update the answer.

3. If there will be no negative edge after all the changes, one can prove that it always makes sense to put all the updates to a single edge. Try each edge uv, calculate the minimum required cost of update if this edge has to be used in the optimal path exactly once. The shortest path from s to t that uses edge uv can be computed as $d = \min(\rho(s,u) + w(uv) + \rho(v,t), \rho(s,v) + w(uv) + \rho(u,t))$. You then need to decrease by $\max(0, a + d - b)$, this will cost $\max(0, a + d - b) \cdot c(uv)$.

4. To run step 3 efficiently you need to compute the shortest paths from s to each other node of the graph, as well as the shortest paths from each node of the graph to t. That can be done by running Dijkstra's algorithm twice.