Code Jam 2017 - Round 2

Analysis: Shoot the Turrets

Shoot the Turrets: Analysis

Small dataset

The first thing to notice is that, if a given soldier s is not the first to perform her task, there is no reason for s to move at all until all soldiers that are to shoot a turret before s have finished.

Let us call the output a strategy: a pairing between some soldiers and all turrets, in a specific order. Given a fixed strategy, we can check if it's viable with one BFS per soldier, in order. We stop with the first soldier that can't reach its assigned turret, if any, and report the number shot so far. Any strategy that shoots a maximum number of turrets can be extended to a pairing of all soldiers to turrets, so this covers all possibilities. Besides the usual empty spaces and walls, we need to consider some empty spaces enterable but not exitable: the spaces that are in the line of sight of turrets that are to be destroyed by a soldier after the current one. Each check takes time linear in the size of the map, so O(**RC**) per soldier.

Unfortunately, there can be too many strategies to bruteforce them. There can be up to 10! possible pairings and 10! orderings, which means $10!^2$ strategies, which is way too big, so we need to do better. We will use precomputing to optimize the check and dynamic programming to optimize the strategy generation.

Let S be the number of soldiers and T the number of turrets. To make the check faster, we can compute, for each soldier and each possible subset of still-alive turrets, the set of reachable turrets to shoot. That requires at most $2^T \times S$ BFSes like the ones mentioned above, for a total complexity of $O(2^T \times S\mathbf{RC})$, which is fine for the Small limits.

Now on to the dynamic programming: consider the function f(s, t) which finds a strategy that shoots a maximum number of turrets for a given set of remaining soldiers s and a given set of remaining turrets t, or decides that there isn't one. Each of s and t can be represented with a binary string of up to S and T digits, respectively. The domain of the function, then, is of size up to 2^{S+T}. To compute the function, we can check every soldier against every reachable turret. Since that list is precomputed, this takes at most S iterations over lists of length up to T. The overall computation of f, if memoized, has a complexity of $O(2^{S+T} \times ST)$, which again, finishes comfortably in time under the Small limts.

Large dataset

For the Large dataset, anything with exponential complexity seems doomed to fail, so we need to go a different route. We will reuse the the BFS idea (to check out whether turrets are reachable) from the Small, but there are a lot of additional insights.

Let us build a bipartite graph G with a node for each soldier and turret. G has an edge from soldier s to turret t if and only if soldier s can destroy turret t after all other turrets have been destroyed. A single BFS starting from each soldier can build this graph. We state the following: destroying R turrets is possible if and only if there is a matching in G that covers R turrets. The only if part is trivial, as the conditions for edges in the graph are a relaxation of the conditions for the pairing we need to construct as solution. We concentrate on proving the if part. Moreover, we provide here a computably constructive proof, for which there is an efficient enough algorithm, meaning the proof is also an algorithm that solves the problem.

First, if G is empty, the problem is solvable trivially by vacuity. For non-empty G, we can use the Ford-Fulkerson algorithm to find an initial maximum matching M of G of size R. We will further consider only the soldiers present in M and ignore the others. From now on, we refer only to soldiers matched by M, and we remove the unmatched soldier nodes from G. Let us define the graph G' with the same nodes as G, but an edge between soldier s and turret t only exists in G' if s can destroy t with the other turrets active. Clearly, G' is a subgraph of G. The outdegree of each soldier node in M is at least 1. If an edge (s, t) in G does not exist in G', it is because some other turret t' is reachable by s and blocking the path to t. But then, (s, t') is in G'. Therefore, the outdegree of each soldier not in G' is at least 1.

Consider the graph H that is the union of G' plus the edges in M reversed. If H contains an edge (s, t') where t' is a turret node not covered by M, let M' be equal to M, but replacing (s, t) by (s, t'), where (s, t) was the edge covering s present in M. Of course, M' is a maximum matching of G of the same size as M. If there is no such edge in H, do the following: since H is a supergraph of G', the outdegree of each soldier node in H is at least 1. The inclusion of the reversed edges of M in H means the outdegree of all turret nodes matched in M in H is at least 1. Therefore, starting at any soldier and moving through edges in H, we will always encounter nodes with outdegree 1 of soldiers and turrets covered by M, and eventually find a cycle of them. Let us call this cycle C. Notice that the edges going out of turrets in H are only the reversed edges from M, so C is necesarily an alternation of reversed edges from M and edges from G'. Consider a new matching M' of G consisting of the edges of M whose reverse is not in C, plus the edges in C whose reverse is not in M. That is, M' is M but exchanging the edges present in C in some direction. M' in this case is also a matching of G of the same size as M. If the cycle is of length 2, then M' ends up being exactly the same as M.

In both cases, we constructed a new matching M' of the same size as M with the additional property that at least one edge of M' is present in G'. Therefore, there are some edges in the matching that represent a number A > 1 of actions that we can take now. So, we can just take those A actions, remove all A used soldier nodes and A destroyed turret nodes, and we are left with a smaller graph which also has a maximum matching of size R - A (whatever is left of M'). Rinse and repeat until the graph is empty.

Complexity analysis. Building the original G takes time O(S**RC**) for S BFSes, and the original matching takes time $O((S+T)^3)$ (there are faster algorithms for matching, but Ford-Fulkerson is more widely known, simpler to code, and sufficient for this task). After that, we have at most T steps of altering the matching and removing some parts of it. Each of this steps requires building G', which takes time O(S**RC**) for S BFSes, and after that, all steps are linear in the size of the graphs G, G' or H, which are all bounded by $O((S+T)^2)$. Notice that building G' is by far the slowest step, so you can use less efficient implementations to manipulate the graphs without altering the time complexity. This gives an overall complexity of $O(TS\mathbf{RC} + (S+T)^3)$. This is enough to solve the problem, but it can be further refined by noticing each time we build G', the BFSes will reach the same places or farther than in the previous step, so, if instead of restarting from scratch we remember the BFSes and continue from where we were stopped before, we can reduce the total time to build G' by a factor of T, down to O(S**RC**), reducing the final complexity to $O(S\mathbf{RC} + (S+T)^3)$.