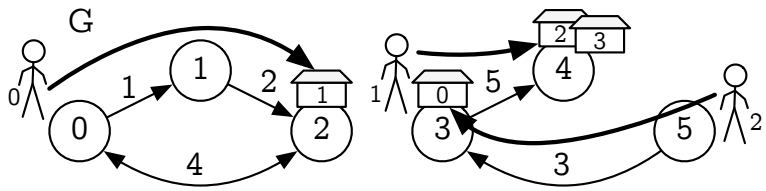
Solution — On Her Majesty's Secret Service

1 The problem in a nutshell

Given a directed graph G with edge travel times, what is the fastest way for a agents to each reach a different one of s shelters?

Once an agent reaches a shelter, he has to wait d seconds before he can enter it. Here is how this looks like for one of the examples on the task sheet:



If $d = 1$ all agents are secure within 6 seconds. Note that even though agent 1 at vertex 3 has a shelter available right there, he takes one for the team and travels to one of the shelters at vertex 4 so shelter 0 can be used by agent 2.

In the input, we are given the description of the graph as well as the lists of agent locations α_i and shelter locations σ_i. For the last 20 points, each shelter will fit two agents if they enter at least d seconds apart, but we will not worry about this for now.

2 Modeling

Building the ski resort graph. The story describes the Schilthorn skiing resort whose structure can be fully described by an abstract graph with the vertices representing the *relevant positions* and the edges representing the *slopes* and *ski lifts*.

Note that no geometrical or geographical information, like latitude, longitude or elevation, is given. There are no guarantees on the structure of the travel times along the slopes and ski lifts. The triangle inequality might not hold, for example. So most likely, we will not need any of the geometrical tools that CGAL has to offer. Since the task, on first glympse, seems to be mostly about travel distances and optimally assigning agents to shelters, the tools from the Boost Graph Library, like Dijkstra's shortest path algorithm, maximum cardinality (bipartite) (weighted) matching and (minimum cost) maximum flow, seem more promising here. But maybe, who knows, we can also find an ad-hoc, STL-only solution?

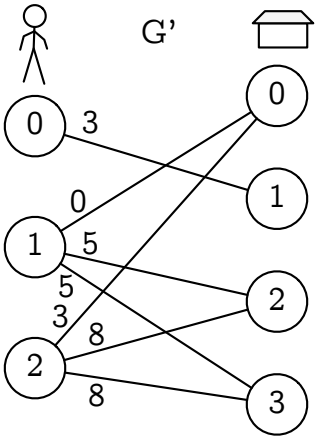
The input is given as a *mixed* graph; a graph that contains both directed edges (skiing slopes) and undirected edges (ski lifts). As there are no limitations on how many agents can travel through each edge at any point in time, we can convert the input to a directed graph, by replacing every undirected edge by two directed edges of the same length, one in each direction. Let $G = (V, E)$ be this edge-weighted, directed graph. Note that there might be multiple edges between the same (ordered) pair of vertices in this graph. If we represent G with BGL's

`adjacency_list`, this is not an issue at all. But if we would use an `adjacency_matrix` (which we almost never recommend you to do) then we can store only one of those edges and so we would have to make sure that we remember the shortest of those edges[1].

Extracting the distances. The task description clearly states that the only point where agents interfere is when they want to enter the same shelter. The travel time for an edge stays the same, no matter how many agents want to use the same slope or lift at the exact same time[2]. So as long as we make sure that each agent travels to a different shelter (for the 80 point subtasks), they can pick their travel routes completely independently from each other. Thus we can say:

If agent i wants to reach shelter j he should take any shortest path from α_i to σ_j.

Given this insight, it is apparent that the only thing we want to extract from our graph G is the value $d_G(\alpha_i, \sigma_j)$, the shortest path length from α_i to σ_j, for every potential assignment of agent i to shelter j. For a single agent (i.e. the first subtask, already worth 20 points), this is already all we need to compute. We just assign the agent to its closest shelter, so the final answer is $\min_j d_G(\alpha_0, \sigma_j) + d$. For multiple agents, we want to collect all the distances and have to do some more work. Let A denote the set of agents ($|A| = a$) and let S denote the set of shelters ($|S| = s$). We build the weighted bipartite graph $G' = (A \cup S, E')$ where (i, j) is in E' if and only if there is a path from α_i to σ_j in G and assign the weight $w((i, j)) = d_G(\alpha_i, \sigma_j)$ to this edge (i, j). Once we have G', we can forget about G. This is how G' looks like for the example from above:



A new kind of matching. What are we looking for in this auxiliary graph G'? We want to associate a single shelter to each agent without using the same shelter for multiple agents. So we want to pick a subset M of the edges in E' such that each agent is incident to *exactly* one edge in M and each shelter is incident to *at most* one edge in M. Thus M should be

[1] We will see in a second why it is ok to forget about the other edges.

[2] This might admittedly be slightly unrealistic, but the task would get much more complicated otherwise. In "reality", one would need to consider travel times that are individual (some agents are better skiiers than others) and also depend on the current time and weather. So no matter how far we go, there is always a point where the tasks in this lab simplify things. It is important for you to recognize this point so that you do not end up trying to solve a problem that is much harder (or much simpler) than what it was meant to be. To quote Albert Einstein: *"Everything should be made as simple as possible, but not simpler."* - although attributing this quote to Einstein might be an oversimplification on its own... (Link)

a *matching*, more specifically an A-saturating matching, meaning that we want $|M| = a$. If there are not enough shelters, for instance, so $s < a$, or if there is a set of k agents who can collectively reach only $k' < k$ many different shelters, such a matching surely will not exist (*Hall's* condition). However, the task description states that *"there always exists a way such that all the agents can reach and enter a shelter"*. So we don't have to worry too much about this, there will always be a matching in G' of size a.

But what is the optimization objective for this matching? In class we have seen minimum cost maximum cardinality bipartite matchings and that they can be efficiently computed using the algorithms for minimum cost maximum flows in BGL. There, the sum of the weights of all the edges selected to be in M is minimized. This would correspond to minimizing the total travel time of all the agents. However, in our task, we do not care whether some agents seek shelter much before the launch of the avalanche (and hence reduce the total sum of travel times) or if all of them just barely make it in time. We only want to minize the *advance warning time*, being an threshold on the *maximum* weight of any edge that is chosen. So instead of minimizing the sum of edge weights, we want to minimize the largest edge weight. We call this a minimum *bottleneck* matching.

3 Algorithm Design

Minimum bottleneck matching. Before we go into how to compute the auxiliary graph G' in the first place, let us continue with where we just left off: with the algorithmic challenge of finding a minimum bottleneck matching.

If, for subtask 2, we want to decide whether a warning time of 9 seconds is enough, we have to try to build M only using edges of weight at most $9-d$ (so that the agents arrive at least d seconds before 9 seconds have passed). We thus define the bottleneck threshold graph $G'_t = (A \cup S, E'_t)$ with $E'_t = \{e \in E' \mid w(e) \leqslant t - d\}$. If there is an A-saturating matching $M \subseteq E'_9$ in G'_9, we know that 9 seconds are enough, otherwise the answer is 10. To check for such a matching M, it is sufficient to completely forget about the weights and compute a maximum cardinality matching on the unweighted graph G'_9.

For subtask 3, we are assured that the answer is at most 10. So the easiest approach is to repeat what we did for subtask 2 and to build all the graphs G'_1, G'_2, ..., G'_9, G'_{10} and try to find a matching M of cardinality a in each of them. The final answer is then the smallest time for which this worked[3].

In subtask 4, there is no constraint on the answer, so it could be as large as the maximum weight in G' plus d. Extending the linear search from subtask 3 will time out. The maximum weight might be as large as $n \cdot \max z = 10^3 \cdot 10^4 = 10^7$. This would result in way more auxiliary graphs to be built and maximum matchings to be computed than we can afford. But please observe that the predicate *"Are t seconds enough?"* that we are computing for each time t has the property of being monotone: If we can do it in t seconds, we can also do it in $t+1$ seconds (by just letting every agent wait for a second at the end). Hence the problem admits a *binary search* for the smallest value t such that G'_t contains a matching of size a[4]. As the

[3] This alone gives you just a 15 point solution, but explicitly combining this with the single agent case from subtask 1 scores 40 points.

[4] The monotonicity can also be seen by the fact that the edge sets E'_{t_1} and E'_{t_2} of two graphs G'_{t_1} and G'_{t_2} are only increasing over time, so $E'_{t_1} \subseteq E'_{t_2}$ if $t_1 \leqslant t_2$.

size of the range $[0, n \cdot \max z + d]$ needed for this binary search is roughly bounded by 10^7 the number of graphs and matchings to be built is sufficiently small: $\log_2 10^7 \leqslant 24$.

All-pairs distance graph. To build G', we need to compute all pairwise distances from all agents to all shelters. The simplest way to do this is to compute all shortest path lengths between any pair of vertices in G (often abbreviated as the *APSP* problem). BGL offers multiple algorithms to solve APSP (none of them were explicitly covered in the class):

- `floyd_warshall_all_pairs_shortest_paths` (Doc) which runs in time $\mathcal{O}(n^3)$, and
- `johnson_all_pairs_shortest_paths` (Doc) which runs in time $\mathcal{O}(n \cdot m \cdot \log n)$.

As our graph G is rather large and rather sparse, the Floyd Warshall algorithm clearly is too slow (n^3 is in the order of 10^9). But also Johnson's algorithm will only score a few points if you manage to keep the overall constant factor fairly small (as $n \cdot m \cdot \log n$ is also in the rough order of 10^8).

In general, these APSP approaches are too slow because they compute much more than what we really want to know. So this is a point where it is crucial to carefully look at the limits given in the task statement. As there are at most 100 agents and 100 shelters (out of 1'000 vertices overall), we are only interested in the at most 10'000 vertex distance pairs (out of $n^2 \leqslant 1'000'000$ many vertex pairs overall). Therefore, it will be roughly ten times faster (both in theory and in practice) if we just compute the shortest distances from every agent's starting position to every vertex of G (including all the shelters). So the algorithmic approach that you should take to compute G' is to run `dijkstra_shortest_paths` (Doc) a times, once with every α_i as the starting vertex. This has the overall time complexity $\mathcal{O}(a(n \log n + m))$.

Handling larger shelter capacities. Finally, we have to talk about how to solve the last subtask. There, two agents can enter the same shelter if they enter at least d seconds apart from each other. While this might sound complicated at first and breaks some of our assumptions above (e.g. that we can assume that $a \leqslant s$), we actually do not need to change much of our approach to incorporate these capacities. For instance, it still holds that every agent can and should travel along a shortest path to his targeted shelter. He can wait in front of the door in case he will be there early while another agent is still in protocol there.

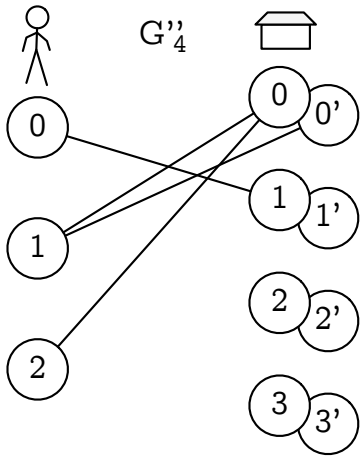
We slightly change the construction of our auxiliary graph in order to reflect that a second agent might get assigned to a shelter. We duplicate every vertex in the shelter partition of G' and also duplicate all the incident edges. Formally, we define $G'' = (A \cup S \cup S', E'')$ with $(i, j) \in E''$ and $(i, j') \in E''$ if and only if there is a path from α_i to σ_j in G and we assign the weight $d_G(\alpha_i, \sigma_j)$ to both edge (i, j) and edge (i, j').

The distinction between the two slots in each shelter occurs in the new threshold graph G''_t:

> In order for an agent to be the *second to last* agent that enters a shelter, he has to arrive at the entrance at least $2d$ seconds before the avalanche is launched.

Thus $G''_t = (A \cup S \cup S', E''_t)$ with $E''_t = \{(i, j) \in E'' \mid w((i, j)) \leqslant t - d\} \cup \{(i, j') \in E'' \mid w((i, j')) \leqslant t - 2d\}$ is the graph where the existence of a matching of size a implies that the agents can all hide within t seconds. The figure below shows G''_4 with $d = 1$ for our example. As you can see, this graph admits a perfect matching and so the advance warning time decreases from 6 to 4

for capacity 2. As in the previous subtasks, a binary search allows us to find the smallest such t.



Analysis. To summarize the asymptotic runtime behavior of our approach:

(1) We spend $\mathcal{O}(n + m + a + s)$ to read and transform the input into graph G.

(2) In time $\mathcal{O}(a \cdot (n \log n + m))$, we compute the necessary distances and build G''.

(3) In each of the $\mathcal{O}(\log(n \cdot \max z))$ iterations of the binary search for the answer, we compute a maximum cardinality matching at cost $\mathcal{O}((s + a) \cdot (s \cdot a) \cdot \alpha(s \cdot a, s + a))$ (Doc).

With the given input constraints, each of those steps has a limit that is within the order of 10^6, so we are ready to start implementing.

4 Implementation

Given that the two central algorithms we need are readily available in BGL, the implementation is rather straightforward. Here are a few things to watch out for:

- Running the `dijkstra_shortest_paths` algorithm multiple times on the same graph but with many different starting points can easily be done using an exterior property map to get a different distance vector for each source.
- Even though we are looking for a matching in a *bipartite* graph, which we can find using maximum flows, we can use `edmonds_maximum_cardinality_matching` algorithm (Doc). That algorithm also works for non-bipartite graphs and is fast enough for our purpose.
- When implementing the binary search, the usual risks apply. Starting with `INT_MAX` as the upper bound is a good choice (taking the maximum agent-to-shelter distance plus $2d$ would be even better), but one has to be careful not to run into an overflow when computing the new query point. While $(l+r)/2$ and $l+(r-l)/2$ are equivalent algebraically, the latter one omits the potential overflow.
- Rebuilding the matching graph G''_t from scratch in every iteration is fast enough. G''_t is relatively small compared to G and we will only build it roughly 30 times, so there is no

need to try to build it incrementally or decrementally by adding or removing edges from the matching graph of the previous iteration.

- The shelter capacity really only comes into play at this one point when we build G_t''. Once you understood the idea of its construction, it should also be clear to you how one could extend it to support arbitrary capacities (at the cost of a graph of size $\mathcal{O}(a \cdot s \cdot c)$).

5 A Complete Solution

```cpp
#include <iostream>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/max_cardinality_matching.hpp>

using namespace boost;
using namespace std;

typedef adjacency_list<vecS, vecS, undirectedS, no_property> Graph;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_weight_t,int>> DiGraph; // An edge-weighted digraph.
typedef property_map<DiGraph, edge_weight_t >::type       WeightMap;
typedef graph_traits<DiGraph>::vertex_descriptor          Vertex;
typedef graph_traits<DiGraph>::edge_descriptor            Edge;

void testcase() {
    // Read the input and build graph G.
    int N, M, A, S, C, D; // Capital letters used for the input sizes.
    cin >> N >> M >> A >> S >> C >> D;

    DiGraph G(N);
    WeightMap weight_map = get(edge_weight, G);

    for (int m = 0; m < M; ++m) { // Lower case letters used for the counters.
        char w; int x, y, z;
        cin >> w >> x >> y >> z;
        Edge e;
        tie(e, tuples::ignore) = add_edge(x,y,G);
        weight_map[e] = z;
        if (w == 'L') { // Add the reverse edge only for ski lifts.
            tie(e, tuples::ignore) = add_edge(y,x,G);
            weight_map[e] = z;
        }
    }

    // Compute one distance  map per agent.
    vector<vector<int> > distmap(A, vector<int>(N));
    for (int a = 0; a < A; ++a) {
        int i; cin >> i;
        dijkstra_shortest_paths(G, i,
            distance_map(make_iterator_property_map(distmap[a].begin(),
                                                    get(vertex_index, G))));
    }

    // Represent G' as the pairwise distance matrix T from agents to shelters.
    vector<vector<int> > T(A, vector<int>(S));
    for (int s = 0; s < S; s++) {
        int j; cin >> j;
```

```
        for(int a = 0; a < A; a++) {
            T[a][s] = distmap[a][j]; // Unreachable pairs are at distance INT_MAX.
        }
    }

    // Binary search for the smallest t.
    int low = 0, high = INT_MAX;
    while (low < high) {
        int mid = low + (high-low)/2;
        // Build the bipartite matching graph G''_t.
        Graph GG(A + C * S);
        for (int a = 0; a < A; ++a) {
            for (int s = 0; s < S; ++s) {
                if (T[a][s] == INT_MAX) continue; // Ignore unreachable shelters.
                for (int c = 0; c < C; ++c) { // c agents will still follow.
                    if (T[a][s] + (c + 1) * D <= mid) { // Agent can enter in time.
                        add_edge(a, A + c * S + s, GG); // Add the edge to G''_t.
                    }
                }
            }
        }
        // Compute the size of the maximum cardinality matching.
        vector<Vertex> matemap(A + C * S);
        edmonds_maximum_cardinality_matching(GG,
            make_iterator_property_map(matemap.begin(), get(vertex_index, GG)));
        const Vertex NULL_VERTEX = graph_traits<Graph>::null_vertex();
        int matching_size = 0;
        for (int a = 0; a < A; ++a) { // Count the number of matched agents.
            matching_size += (matemap[a] != NULL_VERTEX);
        }
        if (matching_size == A) { // The agents all make it in time.
            high = mid;
        } else { // Some agents do not make it, so we need more time.
            low = mid+1;
        }
    }
    cout << low << endl;
}

int main() {
    int T; cin >> T;
    while (T--) testcase();
}
```