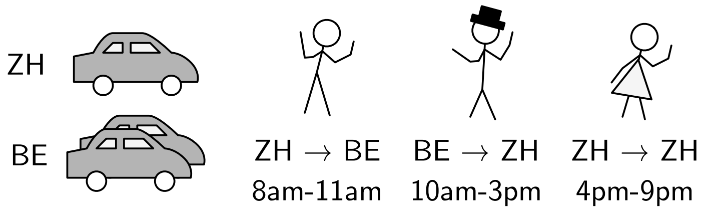
Solution — Carsharing

1 The problem in a nutshell

In this task, we are running a car rental agency. When given S car rental stations, each with their initial number of available cars, and a set of N car rental requests, each with start and end time, start and end location and its profit, our job is to select a feasible subset of requests (one that can be served with the cars available) so that we maximize our total profit.



There are five meaningful subtasks:

(1) There is only a single car.

(2) There are at most 20 requests.

(3) There are up to 10′000 requests.

(4) The request times are no longer multiples of 30 minutes below 10′000.

(5) There are more than just two rental stations.

2 Modeling

Let us go through the list of topics from the course to see what might fit.

- There seems to be little hope that a *greedy strategy* might work. For instance, if we would just accept the requests on a first come first serve basis then we might miss out on a lucrative rental later on. You can easily find counter examples to also all the other strategies that come to mind (consider and accept the requests in decreasing profit order, create an optimal schedule car by car, . . .).
- If we can specify a well-defined subproblem, e.g., looking only at rentals ending before some specific time, then we might figure out a *dynamic programming* approach.
- A *brute force* search through all the subsets of requests will not take us far. Only the second subtask with at most 20 requests seems approachable in this way.
- There is no *geometrical* aspect to this problem.

- Our job is inherently binary. We either accept a rental or not, there is nothing in between. So there is little hope to find an efficient and meaningful formulation as a *linear program*.
- There is no obvious graph problem to see here, but the concept of discrete units of *flow* flowing through a network looks promising. We want to let cars flow through space and time, more specifically in such a way so as to maximize the cost picked up along their rides, which is something we know how to do for *maximum flows* of *minimum cost*. This looks most promising but let us try with a dynamic program first.

Dynamic Programming Approach. If there is only a single car (subtask 1), we can efficiently solve all the subproblems of the following form:

What is the maximum profit achievable if the car ends at station s at time t?

Such a dynamic programming table `DP[s][t]` has to consider $2 \times \lceil 100'000/30 \rceil$ many different states[1], but they can each be computed in constant time:

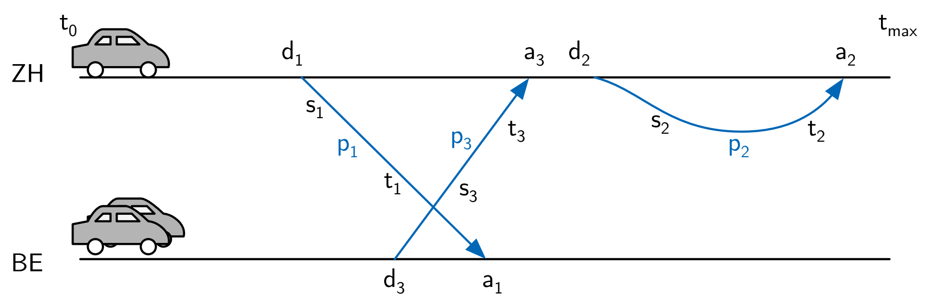
- We initialize with `DP[s][0] = 0` and $\mathsf{DP}[s][t] = \mathsf{DP}[s][t-1]$.
- For all the requests r that *end* at time $30 \cdot t$ at station s, we can either take it or not:

  `DP[s][t] = max(DP[s][t], DP[r.start][r.departure/30] + r.profit)`

If we compute this in increasing time order we respect the dependencies between the subproblems and can read off the final answer at the end of the table as `max(DP[0][T], DP[1][T])` where $T = \lfloor 100'000/30 \rfloor$. This way, we pick the optimal end station for the car.

Unfortunately, this dynamic program does not easily generalize to multiple cars. At any point in time, an arbitrary subset of the cars might be in transit and so a well-defined subproblem for a specific time t would need to fix in its state on which request each car is currently driving on or where it is currently parked. So the necessary state grows exponentially in the number of cars, requests and stations.

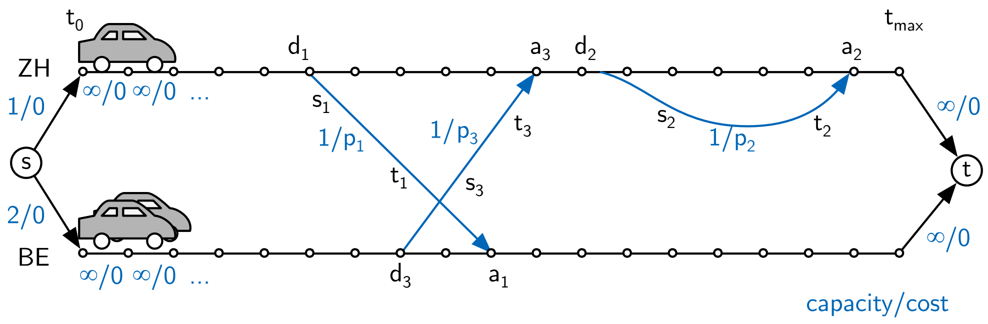
Max Cost Max Flow Formulation. Instead, let us model each car as a unit of flow in the so-called *space-time*-graph. Each vertex of this graph represents a position in time and space, i.e., being at a certain rental station at a certain moment. Being parked is represented by an edge that moves forward in time but not in space. A rental is modelled by an edge that jumps forward in time from the timeline of one station to another (or also potentially the same station).



We can complete the formulation as a *Max*CostMaxFlow problem as follows:

[1]We can divide all the times in this subtask by 30 minutes to get nice integers instead of half-hour multiples.

- We discretize time into the positions where potentially something can happen (half-hour intervals in subtasks (1) to (3), single minutes later on).
- Edges for parking have zero cost and infinite capacity (parking space is unlimited).
- The edge for a rental i has cost p_i and capacity 1 (only a single car is needed to serve the request).
- We add a single super source plus an edge to the beginning of the timeline of each station i which has zero cost and the capacity of l_i, one for each car initially parked there.
- We add a single super sink plus an edge from the end of the timeline of each station i to it with cost zero and infinite capacity (we do not anticipate how many cars will end up there).



There are $L := \sum_{i=1}^{n} l_i$ cars in total. Our flow formulation ensures that the maximum flow from s to t in this graph is always equal to L. Why? The flow is at most L since the cut around the source has capacity L. There is no further bottleneck, since all the cars could just stay parked the entire time, so there is sufficient capacity to let all the cars flow to the sink even without using the request edges. Also because of those uncapacitated parking edges, no car will ever get stuck at a station after taking a request edge. So if we maximize the cost of the maximum flow, we maximize the sum of profits of the served requests (as those are the only edges with non-zero cost) and not necessarily the number of granted rentals. This is exactly what we are after. So the maximum cost among all maximum flows in this graph is our final answer.

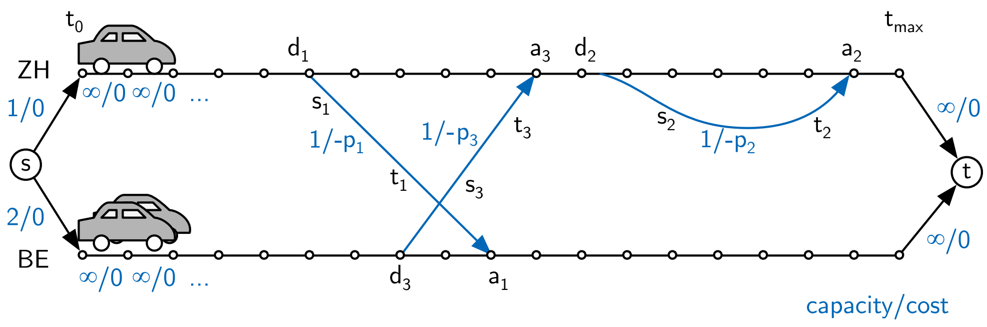
3 Algorithm Design

The get a problem in the *MinCostMaxFlow* format, we can simply negate all the edge costs, so that the request edges get negative costs.

We can apply BGL's `cycle_canceling` algorithm. Its running time is bounded by $\mathcal{O}(n \cdot m \cdot |C|)$, so the product of the number of vertices n, the number of edges m and the absolute value of the cost of the flow $|C|$. In the second subtask[2] with $N \leqslant 20$, we have $n \approx S \cdot T/30 \approx 2 \times 300$, $m \approx n + S + N \approx 600$ and $|C| \approx N \cdot p \leqslant 20 \cdot 100$. So this bound is already in the hundreds of millions, but since the bounds for all the flow algorithms are rather loose, it is fast enough to solve this second subtask.

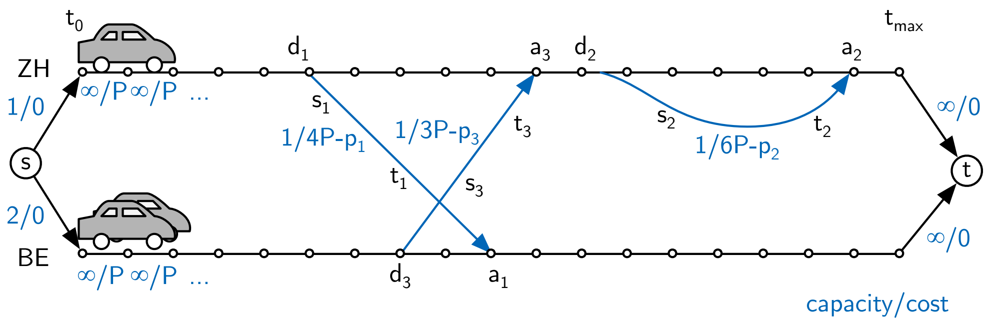
For the further subtasks, we face two main challenges:

[2]Recall that in subtasks 1 to 3 all times are multiples of 30 and below 10'000.

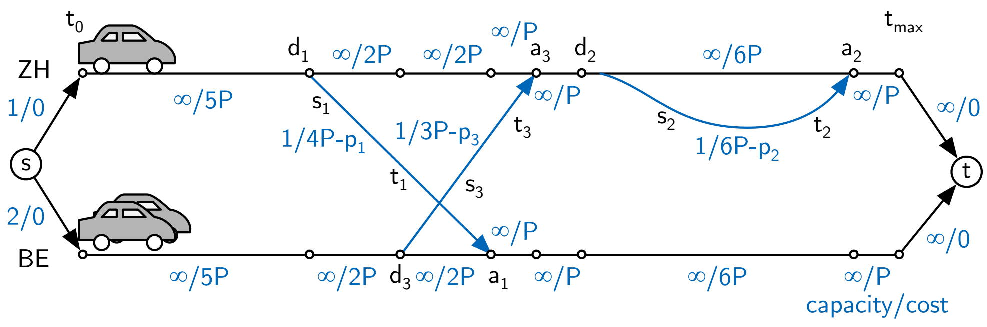


- If the number of requests increases from 20 to 100′000, the cost of the flow grows and with it the running time of `cycle_canceling`. If we want to get rid off the runtime dependency on $|C|$, we have to switch to the other MinCostMaxFlow algorithm `successive_shortest_path_nonnegative_weights` which only depends on n, m and $|f|$, the size of the flow, which is bounded by $L \leqslant 10 \cdot 100 = 10^3$. So to achieve the *good* runtime of $\mathcal{O}(|f| \cdot (m + n \log n))$, we have to get rid of the negative cost edges.
- Once the times are no longer multiples of 30 minutes below 10′000, our discretized space-time-graph explodes into having roughly one million vertices and edges.

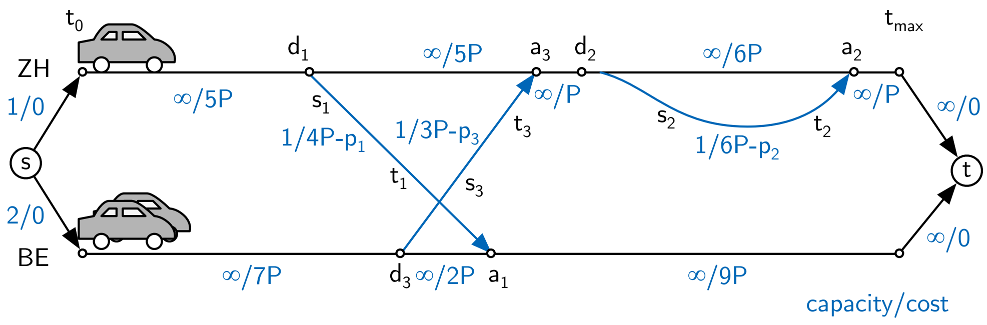
Eliminating negative edges. We address the issue of having negative edge costs with the trick presented in the lecture (for the Real Estate Market problem): We shift the cost of each edge in such a way that each s-t-path gets shifted by the same total amount. By this later property, we can safely compute the offset which we need to deduct in the end. All the edges in our graph move forward in time (except for the few zero-cost edges incident to s and t). So we can define the cost offset with respect to the temporal distance covered by the edge. Let $P = 100$ denote the maximum profit per rental. Each rental covers at least one unit of time (as $d_i < a_i$). Hence, defining the cost offset for an edge from time t_1 to time t_2 as $(t_2 - t_1) \cdot P$ will result in non-negative costs for each edge.



Compressing the graph. As one can sense from the picture above, most of the vertices of our graph are rather boring: They have degree two, meaning that a car that was parked there will stay parked for another time step as there is no rental potentially starting right now. They are not interesting for us. The flow algorithm does not have to decide anything there. These vertices could be eliminated by merging the two edges before and after (summing up their costs, keeping the infinite capacity). But how do we not even create them in the first place? The simplest way to do this is to keep track of all the times occuring in the input. If we only create vertices for those, we can safe roughly a factor of five: at most $20'000$ of the $100'000$ points in time can appear in the input. This simple technique of removing irrelevant positions is sometimes called *coordinate compression* and can also be useful for other applications (e.g., for scanline algorithms, not covered in this course).



But this is not enough yet for the last subtask. As in the picture above, our current construction still creates some vertices of degree two, namely for a station and for a time where a rental starts or ends at *another* station at that time. For two stations this overhead does not matter too much. But now in the last subtask where we have ten stations, doing the coordinate compression *per station* and not globally, really saves us another factor of ten and is a necessary optimization to pass the time limit.



A remark about Casino Royale. In case you are wondering about how to solve the task *Casino Royale*. This can be explained in one line now: Think of the train as a rental agency with a single station ($S = 1$). The seats on the train correspond to the cars and the missions to the rental requests. The coordinate compression is not needed to solve that task.

4 Implementation

With all the necessary speedup insights in place, a complete implementation is straight-forward but still a bit tedious. Here are some tricks for how to simplify your (coding) life:

- Use an `std::set<int>` and/or `std::map<int,int>` for the coordinate compressions of each station. So first insert all the times of the input into a separate set per station and then afterwards put all enumerated times as key value pairs into a map.

- Use the `EdgeAdder` class provided in our templates. This prevents you from writing the code to insert a weighted, capacitated edge more than once and from passing all the property maps over and over again.
- Bundle all five values of a booking into a `struct` so that you can access them as `B[i].s`, `B[i].t` and so on (instead of storing them in separate arrays).

5 A Complete Solution

```cpp
#include <iostream>
#include <set>
#include <map>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/successive_shortest_path_nonnegative_weights.hpp>
#include <boost/graph/find_flow_cost.hpp>
using namespace boost;
using namespace std;

// Graph Type with nested interior edge properties for Cost Flow Algorithms.
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS, no_property,
    property<edge_capacity_t, long,
        property<edge_residual_capacity_t, long,
            property<edge_reverse_t, Traits::edge_descriptor,
                property <edge_weight_t, long> > > > > Graph;
// Interior Property Maps.
typedef property_map<Graph, edge_capacity_t>::type      EdgeCapacityMap;
typedef property_map<Graph, edge_weight_t >::type       EdgeWeightMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type       ReverseEdgeMap;
typedef graph_traits<Graph>::vertex_descriptor          Vertex;
typedef graph_traits<Graph>::edge_descriptor            Edge;
typedef graph_traits<Graph>::out_edge_iterator          OutEdgeIt;


class EdgeAdder {
    Graph &G;
    EdgeCapacityMap &capacitymap;
    EdgeWeightMap &weightmap;
    ReverseEdgeMap  &revedgemap;
public:
    EdgeAdder(Graph & G, EdgeCapacityMap &capacitymap, EdgeWeightMap &weightmap,
        ReverseEdgeMap &revedgemap) : G(G), capacitymap(capacitymap),
        weightmap(weightmap), revedgemap(revedgemap) {}

    void addEdge(int u, int v, long c, long w) {
        Edge e, reverseE;
        tie(e, tuples::ignore) = add_edge(u, v, G);
        tie(reverseE, tuples::ignore) = add_edge(v, u, G);
        capacitymap[e] = c;
        weightmap[e] = w;
        capacitymap[reverseE] = 0;
        weightmap[reverseE] = -w;
        revedgemap[e] = reverseE;
        revedgemap[reverseE] = e;
    }
};

struct Booking {
    int s, t, d, a, p;
};

void testcase() {
    int N, S; cin >> N >> S;
    vector<int> L(S);
    for (int i = 0; i < S; ++i) {
        cin >> L[i];
    }
    const int MAXL = 100;
    const int MAXT = 100000;
    const int MAXP = 100;
    const int INF = MAXL*S; // Infinite capacity.
    vector<Booking> B;
    vector<set<int> > times(S); // Sets for the coordinate compression per station.
    for (int s = 0; s < S; ++s) { // Add sentinels.
        times[s].insert(0);
        times[s].insert(MAXT);
    }
    vector<map<int,int> > M(S); // Mappings for the coordinate compression.

    for (int i = 0; i < N; ++i) {
        int s, t, d, a, p;
        cin >> s >> t >> d >> a >> p;
        s--; t--; // Convert to zero-based times.
        times[s].insert(d); times[t].insert(a);
        B.push_back({s,t,d,a,p});
    }

    vector<int> psum(S+1); // Prefix sums of coordinate lengths.
    for (int s = 0; s < S; ++s) {
        int i = 0; // Create the coordinate mapping for station s.
        for (auto &t : times[s]) {
            M[s][t] = i;
            i++;
        }
        psum[s+1] = psum[s] + M[s].size();
    }

    // Build the graph, setup just like in any other MCMF task.
    int T = psum.back(); // # Vertices = total number of timesteps + 2.
    int v_source = T;
    int v_target = T+1;
    Graph G(T+2);
    EdgeCapacityMap capacitymap = get(edge_capacity, G);
    EdgeWeightMap weightmap = get(edge_weight, G);
    ReverseEdgeMap revedgemap = get(edge_reverse, G);
    ResidualCapacityMap rescapacitymap = get(edge_residual_capacity, G);
    EdgeAdder eaG(G, capacitymap, weightmap, revedgemap);

    for (int i = 0; i < S; ++i) { // Add the edges for the timelines.
        // Source to first station node: L[i] cars, no cost.
        eaG.addEdge(v_source, psum[i], L[i], 0);
        // Last station node to target: arbitrarily many cars.
        eaG.addEdge(psum[i+1]-1, v_target, INF, 0);
        // Add compressed parking edges along each station’s timeline.
        int it = -1, lastt = 0;
        for (auto& t : times[i]) {
            if(it != -1) {
                eaG.addEdge(psum[i]+it, psum[i]+it+1, INF, MAXP*(t-lastt));
            }
            it++; lastt = t;
        }
    }

    for(int i=0; i<N; i++) { // Add the booking edges.
        eaG.addEdge(psum[B[i].s] + M[B[i].s][B[i].d],
                    psum[B[i].t] + M[B[i].t][B[i].a],
                    1, ((B[i].a-B[i].d)*MAXP)-B[i].p);
    }

    // Run the min cost max flow algorithm.
    successive_shortest_path_nonnegative_weights(G, v_source, v_target);
    int flow = 0; // Iterate over all edges leaving the source to sum up the flow.
    OutEdgeIt e, eend;
    for(tie(e, eend) = out_edges(vertex(v_source,G), G); e != eend; ++e) {
        flow += capacitymap[*e] - rescapacitymap[*e];
    }
    // Undo the offset of the flow costs.
    int cost = MAXP*MAXT*flow-find_flow_cost(G);
    cout << cost << endl;
}
int main() {
    int T; cin >> T;
    for(int t = 0; t < T; ++t) {
        testcase();
    }
}
```