Hashcode 2018 — How to secure a global rank of 13 with just 100 simple lines of code and reach World Finals!

▼ Index

HashCode 2018 — Scheduling Self-driving Rides

Abstract

Introduction

We (Nikhil Chandak and Shashwat Goel) decided to attempt Hashcode 2018 in a timed (4-hours) virtual environment, as just a team of 2 as our other teammates weren't available. As the official checker wasn't available as is in-contest, we decided to use the one by PicoJr on Github. We went in hoping to apply some of the discrete optimization techniques we'd recently picked up, specifically Mixed Integer Programming and Local Search. However, with just an easy to code, generalized and extremely intuitive greedy algorithm, we were able to rack up a score that would have us in the top-15 well within the time limit. This was surprising considering the simplicity of our approach, with no *complication* whatsoever. It all depended on one clever one-line selection function. Read on to find out more ;)

▼ Note on the style of blog — it is focused on beginners rather than experienced participants. Toggle to see more details :)

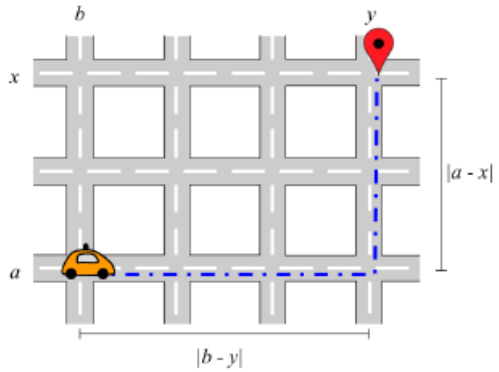
> This blog focuses on the **process** of attempting Hashcode, rather than just providing an editorial of sorts. This is why it's a relatively long post, but we believe it's the process that is more useful when starting off, because the solution to every Hashcode will differ. But some insights remain common to all, which we'll summarize in the conclusion section. Thus, we've proceeded to structure the blog on a temporal basis, adding insights roughly corresponding to when we got them, instead of logically clubbing things together. This leads to a more complicated blog, but a more realistic one. However, for serious future aspirants, hopefully this will be more useful.
>
> For those wishing to quickly check our solution, read sections *Intro to Greedy A*, *Implementing Greedy A*, *The ride picking function*, *200-240 minutes breakthrough*. Our data analysis is in the section "The Data" and conclusions are at the end of the blog.

The problem statement

Luckily, unlike most Hashcodes, the problem statement was short and easy to understand. The scenario is of operating a fleet of self-driving cars on a $R \times C$ grid, initially all positioned at $(0, 0)$. We are given N pre-booked rides signified by **4** key characteristics:

1. Starting coordinates
2. Destination coordinates
3. Earliest possible start time
4. Latest possible finish time

A car can service one ride at a time, but the fleet can operate simultaneously. Each car has to drive to the starting location of the ride to collect it and wait till the earliest possible start time. There is a bonus B for every ride that starts exactly at the earliest possible start time. There is no payment for a ride that ends after the latest possible finish time. Otherwise the payment is proportional to the hamming distance traveled during the ride $= |x_1 - x_2| + |y_1 - y_2|$.



Pretty intuitive right? It almost sounds like Uber, except that here we have the luxury of pre-booked rides and don't have to service them online. Other simplifying assumptions in this comparison include uniform time taken per distance traveled and also uniform payments. The bonus on earliest start time and absolute penalty on crossing the finish time are different from a realistic model.

File format

The first line of the input file contains the following integer numbers separated by single spaces:

- ***R*** – number of rows of the grid $(1 \le R \le 10000)$
- ***C*** – number of columns of the grid $(1 \le C \le 10000)$
- ***F*** – number of vehicles in the fleet $(1 \le F \le 1000)$
- ***N*** – number of rides $(1 \le N \le 10000)$
- ***B*** – per-ride bonus for starting the ride on time $(1 \le B \le 10000)$
- ***T*** – number of steps in the simulation $(1 \le T \le 10^9)$

N subsequent lines of the input file describe the individual rides, from ride 0 to ride $N-1$. Each line contains the following integer numbers separated by single spaces:

- ***a*** – the row of the start intersection $(0 \le a < R)$
- ***b*** – the column of the start intersection $(0 \le b < C)$
- ***x*** – the row of the finish intersection $(0 \le x < R)$
- ***y*** – the column of the finish intersection $(0 \le y < C)$
- ***s*** – the earliest start $(0 \le s < T)$
- ***f*** – the latest finish $(0 \le f \le T)$, $(f \ge s + |x-a| + |y-b|)$
  - note that ***f*** can be equal to ***T*** – this makes the latest finish equal to the end of the simulation

Scoring

The output which we have to produce is straightforward. For each vehicle, we have to output the number of rides it has performed (M) and the order in which it performs the rides (R_i). There are some simplifying yet natural conditions like any ride can be assigned to a vehicle at most once and it is allowed to skip rides (this means that it may happen that in some data files it is not possible to schedule all the rides!).

The checker just simulates all the assigned rides to a vehicle. After finishing a ride, the vehicle is instantly transported to the starting location of the next ride. After arriving, if the earliest start time for beginning the ride is more the current time, it simply waits; otherwise, no need to wait. The checker rewards B bonus points if the ride began exactly at the starting time. Also the (manhattan) distance for traveling is awarded only if the ride finished within the latest time; otherwise, 0 points even if the vehicle completed the ride 1 time unit later. So it's natural to deduce that if the ride cannot be completed on time, then there is no point in scheduling it and we should skip that ride with the hope to begin the next ride on time (thus, getting a bonus).

▼ Click to see the sample explanation for test file A, in case anything is unclear.

`3 4 2 3 2 10`
`0 0 1 3 2 9`
`1 2 1 0 0 9`
`2 0 2 2 0 9`
	`3 rows, 4 columns, 2 vehicles, 3 rides, 2 bonus and 10 steps`
	`ride from [0, 0] to [1, 3], earliest start 2, latest finish 9`
	`ride from [1, 2] to [1, 0], earliest start 0, latest finish 9`
	`ride from [2, 0] to [2, 2], earliest start 0, latest finish 9`

Example input file.

`1 0` `2 2 1`	`this vehicle is assigned 1 ride: [0]` `this vehicle is assigned 2 rides: [2, 1]`

Example submission file.

For **example**, with the example input file and the example submission file above, there are two vehicles.

Vehicle 0 handles one ride:

- ride 0, start at step 2, finish at step 6. Earns points: 4 (distance) + 2 (bonus) = 6

Vehicle 1 handle two rides:

- ride 2, start at step 2, finish at step 4. Earns points: 2 (distance) + 0 (no bonus) = 2
- ride 1, start at step 5, finish at step 7. Earns points: 2 (distance) + 0 (no bonus) = 2

The total score for this submission is 6 + 2 + 2 = 10.

The Data

We first read and ensured we understood the problem statement. Following this, we started analyzing the data on what properties it could be exploited. This was a time-taking process, but we had realized with experience that it's essential. The data file names hinted at the properties, and try guessing what they could mean. Unfold the sections below to see what we managed to decipher, mostly within the first half-an-hour:

▼ Test file A - *example*

This was just for sanity-checking and contributed negligibly to the score.

▼ Test file B - *should_be_easy*

Lower constraints are often the easier variant for any problem. This file had a small 800×1000 grid with just 100 vehicles and 300 rides. It had a decent per-ride bonus and a time duration of just 25000.

▼ Test file C - *no_hurry*

As the name suggests, the 'time' factors for each rides — earliest start and latest finish were at 0 and $200,000$ (also the total duration of simulation), respectively. The grid was 3000×2000 (tied at largest with D). Thus, the bonus value became irrelevant as it was just 1 and collecting them at start time 0 would also be improbable. The aim was to maximize the distance of rides undertaken in the simulation.

▼ Test File D - *metropolis*

Initially, we couldn't really spot a property except the small bonus of 2. The grid was 3000×2000 (tied at largest with D), with just 81 vehicles but a whopping $10,000$ rides to serve in a duration of $200,000$. Around halfway through the contest, Nikhil decided to look it up with the hope of finding some new property — *metropolis* meant 'large and busy city' unlike what Shashwat initially thought could be a hint towards the metropolis meta-heuristic in local search. It made perfect sense since the grid is one of the largest and the number of rides to be scheduled is much more than the number of vehicles available. However, apart from understanding the general description of the data, there was no nice property which could be exploited, giving hints it could be just a large *random* test file potentially among the hardest to optimize.

▼ Test file E - *high_bonus*

As you may have guessed it, the earliest start time bonus in the file was high — 1000. At 1500×2000, the grid was medium-sized with 350 vehicles to serve $10,000$ rides as in test D. The total duration was $150,000$.

Our Virtual Contest

The first hour - Exploration, Ideas and Bounds

We began by reading and ensuring we understood the problem statement. This was followed by exploring all the data files and noting our observations about the data, which came iteratively. We did consider doing more detailed analysis of each data file, including the average distances cars have to travel, to get an idea of whether rides took the car from one end of the city to the other or what. However, there's always an inherent laziness to this, no matter how much you wish you could write up a clean insightful visualization of the whole data, which even given infinite time, might be infeasible for such datasets.

Within the first hour, we started discussing whatever greedy approaches came to our mind. It's impossible to document every single approach that came to mind for most were just passing ideas at the back of our minds. However, most were greedy, until our thought process evolved a bit further which we'll detail.

Intro to Greedy A - Simulation with greedy ride picking on-the-go

Initially, Nikhil suggested just using the low duration in test-file B to naively simulate the process, picking a new ride using a to-be-decided heuristic for each vehicle when its previous one got over. The complexity of this process is roughly $O(T + N^2 + NF)$ [more precise calculations in a later section], fast enough for all the cases. However, Nikhil thought this was just too naive (~~spoiler: ha!~~) and wanted to think more, and so we did. For now we'll refer to this approach as Greedy A.

Calculating Upper Bounds

Shashwat initially floated the comparison between scheduling algorithms in Operating Systems and the problem at hand, but Nikhil was smart to stop that chain of thought in its tracks, before time was wasted on further exploration. The key difference was the ***transition time*** between 2 rides, that is shifting a vehicle from one ride to another, and then making it wait till the earliest start time of the latter ride when required. This transition time turns out to be the key factor in this problem, because score `= T*F - sum(transition time) + bonuses`. This is because $T * F$ is the maximum possible score if the cars are serving rides throughout the duration assuming no late finish or waiting. So essentially we have to minimize this transition time. But notice that it's not as simple as computing the transition time matrix as weights between 2 rides and applying TSP, because

1. we have multiple cars
2. transition time from ride A to B changes based on *when* ride A was completed in the first place, because that decides how much time the car wastes on waiting for the earliest start time of B.

A tighter-bound is calculating the sum of distances for all rides for every test-file, and assuming we manage we achieve bonus for each ride. This bound was helpful in setting a benchmark towards what the ***scope*** for optimization on each file is.

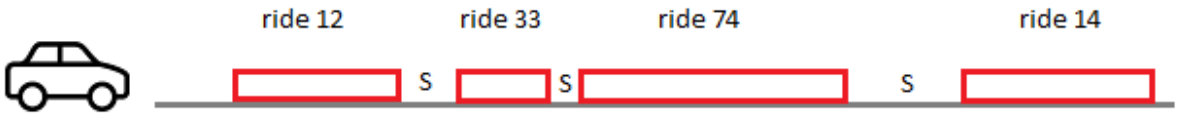
▼ Pro Tip

> If you're within 0.1% of the optimal value in a test file, it is better to focus on other files where there is larger scope for improvement. Well, if you're within 0.1% of the optimal value in ***all*** the test files, then I guess you have already secured a seat in Hashcode Finals? ;)

60-140 minutes - Concrete approaches implemented

Greedy B - Earliest Finishing Ride First for Bonus Maximization (Test E)

Meanwhile Shashwat realized that test E could be approached by just trying to maximize the number of rides scheduled with a bonus, ignoring distances for the time being. Nikhil brought in the comparison with the classic CP problem of choosing maximal non-intersecting intervals over a given time axis. Basically, just sort the rides by ***end*** time (*why not start time?*), and try to assign a vehicle to each so that we maximize the ones that get a bonus. We'll refer to this as Greedy B. Around the 1 hour mark itself, Nikhil decided to take up the implementation for this.



He ensured that whichever ride a vehicle takes, it should always get a bonus on that. After sorting the rides by the *earliest finish time,* one just had to repeat this simple procedure:

- maintain a global list of remaining rides and iterate over each vehicle
- pick the maximal number of non-intersecting rides
- assign it to the current vehicle and calculate the score
- remove the assigned rides from the global list of remaining ones and move to the next vehicle

Nikhil was able to code this in roughly 20-25 minutes and it got a whooping $21M$.

Initial bench-mark of our score

Aa Data Set	# Upper Bound	# Current Score
b_should_be_easy	180798	0

Aa Data Set	# Upper Bound	# Current Score
c_no_hurry	16750973	0
d_metropolis	14272704	0
e_high_bonus	21601343	21338222

It was definitely a good score to begin with but as with every optimization contest, you want more :) Nikhil inspected the output and noticed that $9,877$ out of the total $10K$ rides were successfully taken with bonus (that's quite a lot, we didn't expect to schedule so many rides on time). However, what was even more interesting is that the remaining rides were 123 (unassigned) while the unused vehicles were 174.

So a natural yet extremely simple fix was to go over each un-riden ride and assign a not used vehicle to it, with the hope that the vehicle will be able to complete the ride within the latest finish time. We definitely didn't have scope to get more bonus but we could improve our score by rides' distances/length. This approach improved the score by over $0.1M$ which is quite significant given the difference between optimal and previous score was $< 0.3M$. The score was now $21,465,945$!

We were happy that we got a pretty good score halfway through the contest on E. Nikhil analyzed further on how many rides remained unscheduled and there were 16 of them. Completing them would give an improvement of ~ $13K$. Nikhil thought that since bonus was of utmost priority, it may have happened that vehicles have to wait a lot before taking its next scheduled ride. He felt while waiting there may be sufficient time to try and complete few of the unassigned rides, without bonus of course. He jumped to code it up, without thinking further. But do keep in mind that the improvement at max (with the current techniques) could not be more than $13K$.

General Structure of the Problem

Shashwat meanwhile tried to attack the more general structure of the problem. The observation was that the problem at hand was a mixture of assignment and ordering (routing, similar to traveling salesman). Any solution can be viewed as assigning cars to rides and then ordering the rides of that car for each car independently. At this point, ideas like doing random assignments of cars to rides (as cars were symmetric resources) and then tackling the ordering problem separately, perhaps using Local Search, were considered. However, these seemed time-taking to implement, so more thought and confidence was needed before starting.

Implementing Greedy A

Around 15 minutes (around 75 mins since the start) after Nikhil had started coding the test E greedy, Shashwat decided to implement greedy A. A quick recap, this is just simulating the whole situation by assigning rides to **available** cars greedily on-the-go.

Before implementing, I realized the simulation could be made efficient using a C++ STL priority queue that keeps cars sorted by when they will become (or last became) free. This way we could just check if the top element in the priority queue is already free, and until that condition remains for the queue, pop the top car out, schedule it a new ride, inserting it back.

Shashwat finished implementing before the 2.5 hour mark, and the code came out surprisingly clean and modular. The final code is attached here, and the only significant change from the initial implementation (as at ~135 minutes) is some minor debugging, and changes to the priority score computation for ride picking. Rest are comments for easy understanding :)

▼ C++ Code:

```
#include<bits/stdc++.h>
using namespace std;

int R, C, F, N, B, T;

struct Ride{
    int done, a, b, x, y, earliest_start, lastest_end, length;
}; vector<Ride> rides;

struct Car{
    int x, y, occupied_till;
```

```
    vector<int> served;
}; vector<Car> cars;

struct pqsrt{ //sort in priority queue by till when car is occupied
    bool operator()(const int &u, const int &v){
        return cars[u].occupied_till > cars[v].occupied_till;
    }
};
priority_queue<int, vector<int>, pqsrt> pq; // stores index of car

int distFunc(int x1, int y1, int x2, int y2){
    return abs(x1 - x2) + abs(y1 - y2);
}

void assign(int car_id, int ride_id, int t){
    int transit_dist = distFunc(cars[car_id].x, cars[car_id].y, rides[ride_id].a, rides[ride_id].b);
    int start_time = max(rides[ride_id].earliest_start, t + transit_dist - 1); //max of earliest start time or car reaching the

    cars[car_id].occupied_till = start_time + rides[ride_id].length; //car is now occupied till end of this ride
    cars[car_id].x = rides[ride_id].x; cars[car_id].y = rides[ride_id].y;
    cars[car_id].served.push_back(ride_id);

    rides[ride_id].done = 1;
}

void init(){
    cin >> R >> C >> F >> N >> B >> T; rides.resize(N); cars.resize(F);
    for(int i = 0; i < N; i++){
        cin >>rides[i].a >> rides[i].b >> rides[i].x >> rides[i].y >> rides[i].earliest_start >> rides[i].lastest_end;
        rides[i].done = 0; rides[i].length = distFunc(rides[i].a, rides[i].b, rides[i].x, rides[i].y);
    }
    for(int i = 0; i < F; i++){
        cars[i].x = cars[i].y = cars[i].occupied_till = 0;
        pq.push(i);
    }
}

int choose(int car_id, int t){
    int bidx = -1 ; double bscore = INT_MIN;
    //iterate over rides and pick one with best picking score
    for(int i = 0; i < N; i++){
        if(rides[i].done==1) continue;

        int transit_dist = distFunc(cars[car_id].x, cars[car_id].y, rides[i].a, rides[i].b);
        int start_time = max(rides[i].earliest_start, t + transit_dist - 1);
        if(start_time + rides[i].length > rides[i].lastest_end) continue; // will finish too late so just ignore

        int wasted_time = start_time - t; //time in which car does not earn any score

        int bonus = 0;
        if(start_time == rides[i].earliest_start)
            bonus = B;

        double score = bonus + 0.0*rides[i].length - wasted_time ;
        // double score = bonus + 0.0065*rides[i].length - wasted_time ; //Test C
        // double score = bonus + 0.0*rides[i].length - wasted_time - 0.018*(rides[i].lastest_end - t); // Test D

        if(score > bscore){
            bidx = i;
            bscore = score;
        }
    }
    return bidx;
}

signed main(int argc, char* argv[]){
    freopen(argv[1], "r", stdin); freopen("output_file.out", "w", stdout);
    init();

    for(int i = 0; i < T; i++){ //simulate each time step
        while(pq.size() && cars[pq.top()].occupied_till <= i){ //pick a car that is now free/available
            int car_id = pq.top(); pq.pop();
            int ride_id = choose(car_id, i); //choose a ride for this car
            if(ride_id != -1){ //if a ride exists
                assign(car_id, ride_id, i); //assign this car the ride
                pq.push(car_id); //push car back into the busy queue
            }
        }
    }
    for(int i = 0; i < F; i++){ //output stuff
        int m = cars[i].served.size();
```

```
        cout << m << " ";
        for(int j = 0; j < m; j++){
            cout << cars[i].served[j];
            if(j < m-1) cout << " ";
        }
        if(i < F-1) cout << endl;
    }
}
```

The ride picking function

The only detail left to be decided in this paradigm is how to pick among the various possible next rides for a particular car. I thought the general approach is to pick using a function of the following parameters:

1. Bonus attained on picking next
2. Length of the ride
3. Wasted time on transitioning to this ride

Then each not-previously-done ride could be evaluated using this function whenever we're picking a ride. That leads to $O(N)$ each time we decide to pick a ride, so $O(N^2)$ overall, because on each attempt to pick a ride exactly one gets picked. This is fine considering the test files had $N \leq 10,000$ rides.

Here, it seemed viable to keep the function linear with coefficient multipliers for a) b) and c). This is because intuitively their effect on final score combines linearly. More complicated functions can be tried, but we didn't do this.

1. More bonus attained leads to a proportional (constant 1) growth in score.
2. Longer rides mean less time wasted in transition overall, but also less rides get scheduled leading to less bonus. This one can both have a negative and positive coefficient as its net effect is unclear. So this had to be empirically tuned.
3. An increase in wasted time leads to a proportional drop in score (constant -1). This was still experimented with but the -1 coefficient turned out to be sufficient.

Initial Runs

After implementing, the first run itself gave us a seemingly good score. We got $176,877$ on Test B, ~ $15M$ on test C, ~ $11M$ on test D, and $21,465,945$ on test E. A comparison with the in-contest official scoreboard puts this at ~ *Rank 150-200*. The priority score function for ride picking was simply `bonus + 0.0*rides[i].length - wasted_time` at this point. [Yes, ride length had coefficient 0 initially].

Meanwhile, Nikhil had finished implementing his approach of taking unassigned rides while waiting. Alas, there was no improvement at all in E!

Anyway, we had racked up a pretty good rank on the scoreboard! Shashwat's approach was particularly great because Greedy A had managed to match the score of Greedy B on Test E, something greedy B had been specialized for. This gave us further hints that improving on Test E further might just be very hard, perhaps this was the optimum? We were anyway close to the theoretical upper-bound. Keeping all this in mind, we decided not to work further on Greedy B as Greedy A was promising across the test-sets. Comparing with our bounds, it was clear the scope for major improvement in score lay in test C and D. Especially test D (metropolis) where we were off by almost $3M$.

So overall, we were pretty hyped up and with more than 1.5 hour left, perhaps some smart tuning could take us to the top?...

▼ Before moving on: toggle to view some mistakes we made till here in case you're interested

For E, remember the scope for improvement? Yes, $13K$ in a total of $21M$ - way less than 0.1% of the optimal. Given how narrow the scope was for further optimization, it would have been way better for Nikhil to switch over to other data files for optimization.

As for Shashwat, he decided to write code from scratch as the I/O seemed uncomplicated. But in hindsight, **always** take helper functions for input, output the structures used in the program and other common processing functions like calculating hamming distance between 2 intersections from your teammate if they started coding earlier (as Nikhil had, in our case). It took Shashwat 20-30 minutes to finish writing and testing these generic bits, time which could've

comfortably been saved. Moreover, the structures became very different from Nikhil which could've come back to haunt us if we had to combine approaches at some point (luckily, we didn't!).

140-200 Minutes - Roadblock

Tuning doesn't bring the impact we hoped for...

So both of us set out to tune the score function. For test file C, after some manual ternary searching on the *coefficient for ride length* and arriving at 0.0065, the score increased by around ~ $30K$. Note that the effect of the coefficient on the achieved score wasn't exactly bitonic, but was similar enough for the approach to be effective. It was done manually out of laziness, but also because we got to exercise some human intuition on how much to change the values. While this was a significant push, it wasn't exactly what we'd wished. Our total was now within the top 150.

The sad part was that any coefficient tuning on `bonus + 0*rides[i].length - wasted_time` was only worsening the solution on test file D. We noticed that 17% of the rides were not being served in our best solution, and we thought serving more rides by penalizing ride length should've helped, but the fact remained it wasn't. This can probably be explained by the low bonus (2) in the test file.

Nikhil even tried some exponential scoring functions but they all ended up much worse. We'd spent 30 minutes making minor changes to the scoring function, and were starting to lose hope. Perhaps progress on test file D, the busy metropolis, required more sophisticated approaches like Local Search?

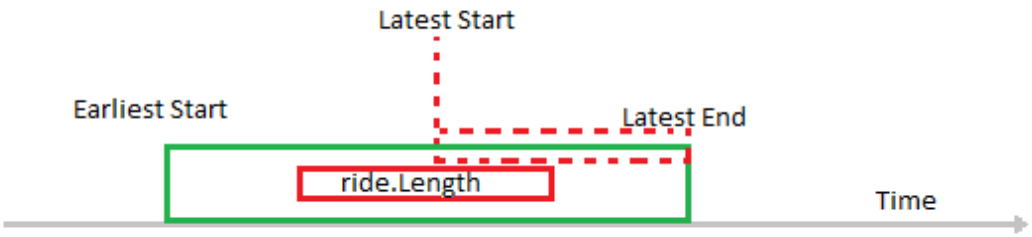
Prospective Local Search Model

We came up with a local search model actually. If we consider a 2D table with each row representing a car, and each row (car) having a subset of rides allocated in order, perhaps the local moves of swapping 2 rides within a particular row (fixed car) or swapping 2 rides across cars would help. Given we had a good greedy initialization, the chances for improvement seemed high. The only issue was implementing a swapping move. A glance at the code would tell you that evaluating swaps in the ordering within a constant factor overhead was non-trivial. There were details like start time and end time that needed to be evaluated. The prospect of accurately coding this within 1 hour seemed daunting, though doable.

Our laziness found itself prevailing, and every minute we procrastinated, hoping for some breakthrough using tuning, the likelihood of implementing local search on time got lesser and lesser.

200-240 Minutes - Breakthrough by augmenting Greedy A picking function

Then came a eureka moment. See if you can spot what information our 3 parameter scoring function could be missing! Hint: Remember that 17% rides were going unscheduled. You might want to check the code linked earlier, it's there...



We weren't incentivizing rides that had an earlier latest finish time! This meant that we gave no priority to rides that won't be available soon overrides which might be available even much later. Bummer! We didn't exactly think this would give a huge boost, but we changed the scoring function to:

```
bonus + 0.0*rides[i].length - wasted_time - coefficient *(rides[i].lastest_end - t ) // t is the current timestep
```

Some tuning on the *coefficient*, and at 0.018 we had broken the $12M$ bar on Test File D! There wasn't a significant increase using this observation on any of the other files though. In any case, this led to our final scores:

Final Score Tally

Aa Dataset File	# Upper bound	# Current Score
b_should_be_easy	180798	176877
c_no_hurry	16750973	15801762
d_metropolis	14272704	12038689
e_high_bonus	21601343	21465945

This meant a total score of: $49,483,273$. A quick glance at the Hashcode 2018 scoreboard would tell you this is enough to place 13^{th}! With such a simple general greedy algorithm and just coefficient tuning for different test files at that! We were even done before our virtual contest duration timed out. The top score was $49,776,211$ and we weren't too far off, even considering the difficulty in improving scores increases exponentially as you get closer to the maximum. Perhaps running our local search formulation using the greedy output as initialization would've taken us there and *maybe* beyond, why don't you try it out and let us know in the comments :)

Conclusion and Key Takeaways

> Small improvement on large gap will be more rewarding then large improvement on small gap, as the final score is sum of individual file scores.

As a consequence, always focus on files that give larger magnitude of output [Eg, ignoring file B in this problem]. For the same reason, having some upper-bounds pre-computed when easy is a nice way to navigate the contest, considering individual file scores are not shown on the leaderboard.

One teammate should write boilerplate code as soon as possible, and make sure everyone sticks to that. This not only saves time for everyone but also makes collaboration and code integration smoother later.

> Keep it simple, stupid i.e. *KISS* principle.

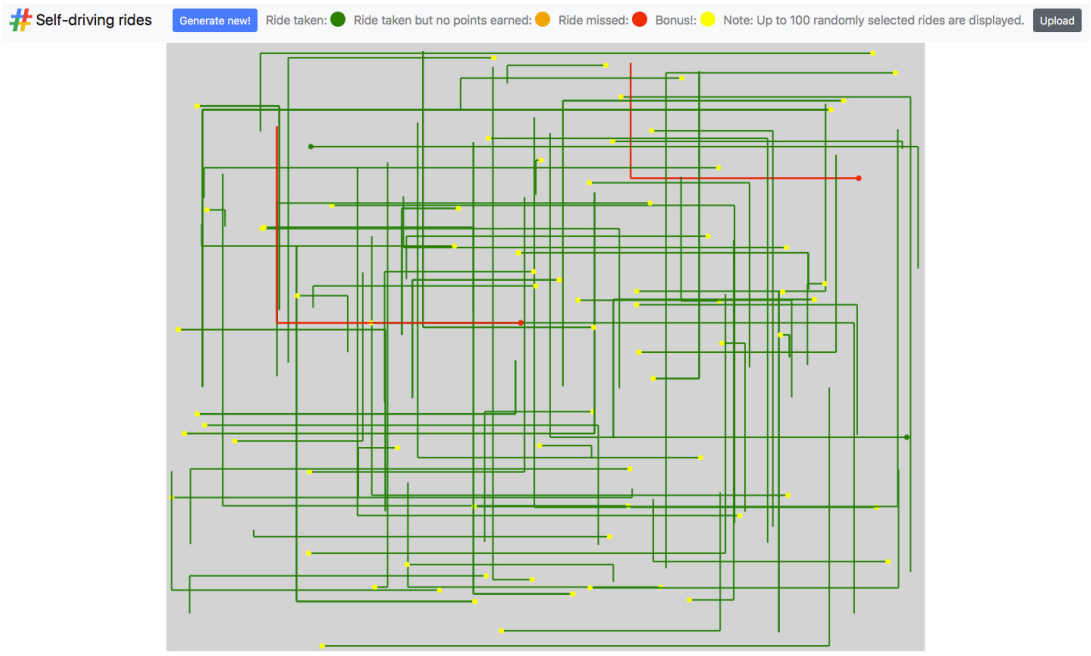
Writing a generic greedy with a selection function that can incorporate multiple parameters will most likely be more useful than complicated techniques like discrete optimization. Something like local search requires a good initialization to be effective mostly, so writing greedy first can virtually hit 2 birds with one arrow.

Improvements like coefficient tuning often bring small gains. They're best done closer to the end, as that's also when we work at our fastest. Moreover, tuning is boring and exhausting simultaneously. In the earlier parts, focus on finding new insights, which might lead to large jumps in score even if simple to code.

> Visualize, if possible.

Apparently, Google had provided a visualizer for Hashcode 2018, which displays the rides scheduled by the code. We had no idea about this before the contest and only got to know about it later. Visualizers can be immensely helpful as it gets harder to manually find properties when the input size increases. Often, just by looking at the output you can see not only how the code has performed but also any distinct feature of the input which you may have missed. If possible, ***always*** use the visualizer. And if you're planning to do some virtual contests like us, do search thoroughly about the resources available before attempting!

Here's a output for dataset B — *should_be_easy* — for you to enjoy!



Doing a few other editions as well (unfortunately not coming up with as effective solutions, hence no blog for these :P), it seems these tips apply across years in Google Hashcode. Hopefully this chronicle of our experience will help you prepare for future HashCode contests. Let us know what you think in the comments! :D

Acknowledgements:

- Simon Galleni's blog on Hashcode 2018 for most of the images used here: http://simostro.synology.me/simone/2018/03/07/google-hashcode-2018/
- Thumbnail from Hash Code 2018 Final Round in Dublin, Ireland - Highlight Reel by Life at Google Youtube Channel.
- Google for providing basically the whole contest — problem statement, test data, etc.
- PicoJr for the checker. Hashcode 2019 & 2020 checkers are also available on his repositories by the way!
- Sebastian Brodehl's Github repository on Hashcode 2018 for the image of visualizer.