# Grid Race Racing agent - Solution of Márton Kozma (DILUOX)

## Table of Contents

This repository contains three solutions to a university project grid race problem. The grid race had three different tiers, each folder contains the bot developed to that specific tier. The main agent is developed continuously through the tiers, therefore the bots are similar, yet they have some differences, deprecated/removed functions, features etc...

# Tier 1 bot

## Describe the problem to be solved.

In the grid race tier 1 we had to implement a simple path finding agent, given the full map that contained wall cells, start cells and goal cells(and traversable cells obviously).

Before starting the race, each agent receive an initial observation from the judge that described the starting position of the agent and the 2 dimension map the agents will race on.

In each round, the agents receive an observation from the judge, which contains the actual position of each agent, and the current velocity of the **receiver** agent.

On the 2d grid, each agent can move in 8 + 1 different positions with the maximum acceleration 1 in both dimensions each round. For example, the agent can output (1,1), which means the racer wants to go one cell west on the vertical axis, and 1 cell south on the horizontal axis, but cannot output anything else which is not in the range of (-1,1).

The abstract task was to develop a bot that can reach the goal position in the least possible number

of steps.

# Are you building on prior work? If yes, cite all sources properly.

I mainly implemented the agent and the path finding algorithms by myself, however, to implement some utilities I used existing source code. I also used external sources describing the desired algorithms, different practices and heuristics.

The following list contains all the used sources and their purpose:

- Source code:
  - Liutenant crown bot(included with the environment): I used the source code which contained an algorithm to detect walls in a straight line between two points on the map.
  - Naive A-star bot(included with the environment): Inspiration of an existing A* algorithm. I did not copy any of the source code from here, but I used it to find bugs, and get an overview of how an a* implemented in python should look like.
- External sources:
  - GeeksForGeeks - Breadth first search: The main idea of bfs.
  - GeeksForGeeks - A*: An overview of a*, and different heuristics that can be used.
  - arXiv:1812.02746; Aniruddha Bapat, Stephen Jordan - Bang-bang control as a design principle for classical and quantum optimization algorithms : The idea of bang-bang control. This idea is used in the heuristic function.

# What representation did you choose for the state space, and why? Would any other representation be viable?

**The state space is represented by objects. There are 3 different objects which represents the state space:**

- Class Track: Reads and stores the grid in a 2D array and provides utility functions to get important information concerning the track(e.g: traversable cells, valid line between two coordinates etc..)
- Class RaceCar: Reads, stores and provides the actual position and velocity of the agent.
- Class EnemyRacer: Reads and stores the actual position of other existing players in the race.

I chose this representation because the object oriented approach makes the program easily maintainable and more readable. The representation of the map is a simple 2D array because it is simple, and efficient.

Other representations could be viable, but may contain more complexities than necessary(e.g: Storing the map as a vector field, calculating and storing the actual velocity of other players etc...).

# Are there uncertainties in the problem?

In tier 1 there are not many uncertainties except one. Where will other players go? We can make stochastic predictions about their next move but it still remains an uncertainty. If the agent chose the same field with the other agent, there could be penalties that effect the overall result of the agent.

# What form of input and output is expected? Are these realistic for a real-word, physical application? If not, is the transformation feasible? (If the transformation requires more intelligence, than solving the problem, then the chosen representation is not really useful.)

- Expected input:
  - Initial observation: three integers separated by spaces regarding the map size(N,M) and the number of agents(K) racing(respectively). After that N lines containing M integers in each line separated by spaces.
  - Observation in each round: Four integers separated by spaces containing the horizontal and vertical position and velocity of the agent respectively. After that, K lines containing two integers in each line separated by spaces - the current position of each agent (Including our own).
- Expected output:
  - Before starting the race, the agent must output 'READY'.
  - Each round: two integers separated by spaces and in the range of (-1,1). This two integers are representing the current velocity change, decided by the agent.
- I personally think that the current representation of the state space may be applicable to real world use. For example, a 3d lidar point cloud can be transformed to a 2d grid, where the agent can find the desired goal. Implementing such transformations that keeps the real life measures are possible, however the current setup should be modified to use floats as coordinates and handle noise. For more complex tasks(like navigate in traffic) this representation is not feasible.

# What algorithms did you consider and which ones did you include in your submission? How do they compare in terms of performance?

I considered the following two algorithms:

- Breadth first search(BFS)
- A*

And implemented the following versions(included in the solution):

- Simple A* with speed limit: While making possible to calculate path with dynamic speed, this algorithm doesn't store anything while finding the solution except the position of each node. Therefore it computes the solution fast, but the computed path may not be optimal.
- A* with velocity as a state: This modified a* stores each node not just by their position, but also the different velocities possible. Therefore, the speed limit is a must, because it finds the optimal path, but due to the lack of optimization, this method have approximately a complexity of O(N^8), where N is the number of nodes in the map. Due to these performance issues, I removed the use of this algorithm in the final solution. For historical and fun reasons I left the implementation in the final submission.
- Dynamic speed BFS: A simple BFS implementation which also considers velocity in each state. This had better performance than the A* with velocity but still, it computed the result relatively slow. Because of that, this algorithm is used in parallel with the simple A* with a timeout. If the function doesn't return a solution in a given time(under 0.9 seconds) we interrupt it and go with result of the simple A*

## Did you implement any AI algorithm by yourself? If yes, then explain what you learned during the implementation work. If not, then describe the aspects you considered when choosing an existing implementation.

Yes, I implemented the A* and the BFS algorithms by myself. During implementation I have learned the importance of an admissible heuristics and also the fact that such a simple algorithm as bfs can take hours to be implemented correctly. I also learned the importance of the order of constraints because if a set of compute heavy constraint evaluations are not executed in the right order, than this can lead to very serious performance issues and compute times. Overall, I earned much confidence and experience in solving algorithmic problems.

## Have you used any heuristics or heuristic functions? If yes, then explain what, why and how?

Yes I used two different types of heuristics:

- Euclidean distance: We learned about this heuristic function in class. It is admissible and simple. I used numpy to implement it.
- Euclidean distance with bang-bang control: I learned about this idea in the cited paper above. The main idea is that we multiply the eucledian distance towards the goal by two, assuming that in the first half of the path, we only accelerate and in the second half, we only decelerate. I thought that under the mentioned circumstances, this heuristic will be more accurate, because it gives an estimate of rounds considering the need of deceleration. After the tests, it turned out I was right, so I kept it. I implemented this heuristic using python's math library.

## Evaluate the performance of the proposed solution, and provide a description of its limitations, and how it could be improved.

On fast processors, the provided solution produces the optimal result(Tested on a Ryzen 7 9700X). However the computational needs of this solution is far beyond optimal and has many limitations. With very large maps, there isn't much chance that the provided path will be optimal. Also, the runtime environment should support multithreading so the algorithm won't work on older systems.

The solution could be heavily improved by optimizing the current pathfinder algorithms. Also the overall solution could be simplified (No need for many different pathfinder algorithms). The most direct way would be to not calculate the full path in each round, just a part of it. We could search for a path not on the whole map, but in its subregions. The resulting solution would have much greater complexity and would return the optimal path without the need of heavy computations. Or it should be implemented in C.

# Tier 2

## Describe the problem to be solved.

The task was very similar to tier 1 except the agents now received only a part of the map, within the visibility radius.

## Are you building on prior work? If yes, cite all sources properly.

No, I developed my tier 1 bot to match the needs of tier 2. All ideas and implementations are my work except for one function:

- From lieutenant crown bot(Included with the environment) I used some parts of the observation reading which considers saving the track. Then I modified this algorithm to only override the previously unseen regions of the map.

## What representation did you choose for the state space, and why? Would any other representation be viable?

It remained the same as in tier 1, except in Class Track, we store the visibility radius too. In this tier, I think there aren't many other viable representations, because we need to track the visited parts of the map.

## Are there uncertainties in the problem?

Yes, many. Where is the goal? Where should we start to go? What is the safest maximum velocity? Is there a dead end where we are going? And there are many other uncertainties in this particular tier.

## What form of input and output is expected? Are these realistic for a real-word, physical application? If not, is the transformation feasible? (If the transformation requires more intelligence than solving the problem, then the chosen representation is not really useful.)

Very similar to tier 1 except a few differences:

- Expected input:
  - Initial observation: Four integers separated by spaces regarding the map size(N,M) and the number of agents(K) racing **and the visibility radius®** respectively.
  - Observation in each round: Four integers separated by spaces containing the horizontal and vertical position and velocity of the agent respectively. After that, K lines containing two integers in each line separated by spaces - the current position of each agent (Including our own). **After that N lines containing M integers in each line separated by spaces containing the map.**
- Expected output:
  - Before starting the race, the agent must output 'READY'.
  - Each round: two integers separated by spaces and in the range of (-1,1). This two integers are representing the current velocity change, decided by the agent.

## What algorithms did you consider and which ones did you include in your submission? How do they compare in terms of performance?

I considered two different approaches to find alternative goals in the visibility limited environment. These are the following:

- Determining the goal cell if it has an unknown neighbor cell, we don’t see any reachable goal cells and we haven’t been on that particular goal cell: This method gave very consistent and robust results, but reached the goal in many steps(200-300 steps on small maps!)
- Determining the goal cell by the most unknown neighbors: This alternative navigation algorithm proved to be the best. Determining the subgoal by the number of unknown neighbors requires a lot of constraints to exclude deadlock scenarios. The resulting implementation performed very well on most of the maps, however, due to some bugs, it isn’t as robust as the single unknown cell approach.

In this tier, I didn't consider any new algorithms regarding path finding. I fine-tuned and optimized the previous ones instead. In this tier, I only used the simple A* algorithm described in the tier 1 section with heavy optimization. This made possible the fact that the overall pathfinder implementation finds a path very fast and the agent have excellent response times.

## Did you implement any AI algorithm by yourself? If yes, then explain what you learned during the implementation work. If not, then describe the aspects you considered when choosing an existing implementation.

Yes, I implemented two methods that determine where should the agent go. Then the agent uses A* to reach that goal. During implementation I learned a lot about fundamentals in navigation under limited visibility conditions. Also (seeing the final result) I learned that robustness is worth more than the number of steps.

## Have you used any heuristics or heuristic functions? If yes, then explain what, why and how?

Yes, I used the same heuristics as I used in tier 1.

## Evaluate the performance of the proposed solution, and provide a description of its limitations, and how it could be improved.

The performance of my tier 2 agent was almost optimal, however it contained some overfitting features which produced worse robustness. Therefore, in the final run my bot competed much more worse than the rest of the competition compared to the fact that my bot was the fastest on the running leaderboards. So it is quite an overfitted agent and in some edge cases, it stuck in an endless loop, running back and forth between two points.

Great improvements could be reached with the removal of the logic which causes the overfitting and fixing the bugs which make the bot useless in some edge cases.

# Tier 3

## Describe the problem to be solved.

The problem was the same as in tier 2 except the agents must navigate on a track which contains oil cells and sand cells which can cause random velocity or slow down the player.

## Are you building on prior work? If yes, cite all sources properly.

No I developed my tier 2 bot to match tier 3.

## What representation did you choose for the state space, and why? Would any other representation be viable?

The same as in tier 2 but Class Track have different cell types as constants.

## Are there uncertainties in the problem?

Yes, many. Most of them are the same as in tier 2, but now we have uncertainties regarding the special cell types too. Should we go on a shortcut filled with sand cells and trade the velocity penalty to a shorter path? What do we do if we accidentally move on an oil cell? These are all uncertainties but there is many more edge cases that we can think of.

## What form of input and output is expected? Are these realistic for a real-word, physical application? If not, is the transformation feasible? (If the transformation requires more intelligence than solving the problem, then the chosen representation is not really useful.)

The same as in tier 2 except in the roundly observation we receive the special cell types too.

## What algorithms did you consider and which ones did you include in your submission? How do they compare in terms of performance?

I included the same algorithms that I used in tier 2 but I removed the lines that caused the overfitting. There isn't anything new except a modified heuristics.

## Did you implement any AI algorithm by yourself? If yes, then explain what you learned during the implementation work. If not, then describe the aspects you considered when choosing an existing implementation.

Yes, I modified my heuristic function by adding a priority to the estimation which is dependent on the cell type. From this I learned that in particular cases like in tier 3, a partly not admissible heuristic can be useful. I also implemented a backup mechanism which activates if the a* won't return a path. This mechanism tries to stay on the path that have been previously calculated. If it's not possible for some reason, then it choses a move which leads to a neighboring cell with the best heuristics that is currently available. This made the agent far more robust than before.

## Have you used any heuristics or heuristic functions? If yes, then explain what, why and how?

Yes, firstly I used the previously written bang-bang euclidean heuristics with an extra priority added to the estimation. If the observed cell is a sand cell we add +30 to the final estimation. If it is an oil cell, we add +50 to the final estimation. This way, we penalize the use of the special cells, but if the solution leads through such cells, the solution is still possible.

## Evaluate the performance of the proposed solution, and provide a description of its limitations, and how it could be improved.

I think that in tier 3, I achieved the best performance with the least number of hours in development. Currently I have the fastest bot on the leaderboard, and my bot achieves great performance on different maps too. It surely can be improved. For example, trying out different heuristics could be great. Also improvements on the current filters in the subgoal selection can make the multi agent race more reliable.