Museum Protection

Nima Iji Edinburgh, UK
Email: ijinima@gmail.com

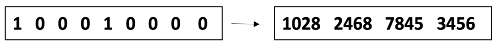


Fig. 1: Binary representation to custom representation.

***Abstract*—This paper addresses inefficiencies in a camera placement optimization algorithm. The initial code struggled to find optimal solutions, prompting modifications to the solution representation and the introduction of an initialization function. Further improvements included a repair function, map boundaries, a new wall variable, distance considerations between cameras, and a custom mutation. These enhancements significantly improved fitness and solution quality across various instances, showcasing the effectiveness of the proposed modifications.**

Keywords: Optimization algorithm, Evolutionary Algorithm

I. Introduction

The default code provided in the resources was remarkably inefficient. Despite imposing substantial penalties, the algorithm persisted in its quest to identify the optimal number of cameras. In numerous iterations, the best fitness reached the maximum attainable fitness, rendering it challenging to determine a clear stopping point for the algorithm. This inefficiency called for a reevaluation of the code to improve its performance and resource utilization.

II. First Modifications

For starting the experiments, first we need to reach a proper minimum solution and then make it better incrementally. For enhancing the current algorithm, using an initialization function and custom representation can be helpful.

A. Representation

Representations in Evolutionary Algorithms are crucial as they significantly influence how solutions are structured and explored, directly impacting the algorithm's efficiency and effectiveness in solving complex problems. They can be helpful to reduce the amount of computation during iterating process of the algorithm. For example, in this coursework we can change the binary representation to a custom representation by using the exact place of cameras. You can see the example in Fig. 1.

B. Initialization Function

In the initial stages of the Initialization function, the primary objective is to commence the process with individuals that possess validity. This involves ensuring that the randomly generated values for each gene do not place the individuals on the walls. To achieve this, a thorough examination is conducted to verify that the assigned random values fall within acceptable parameters, preventing the placement of individuals in prohibited areas such as the walls.

III. First Experiment

After changing the default representation and adding an initialization function, the result got better but not that much. In addition, for doing the experiment we had to deactivate the crossover function because it doesn't have any usage here. The results for the fitness got improved in the fitness factor, but not in the real solution in the map visualization.

Min Fitness	2479
Number of cameras	5
Wall time	11.5s

TABLE I: results for the WallsTest1_5_cameras instance after first modifications

IV. Better results with new approaches

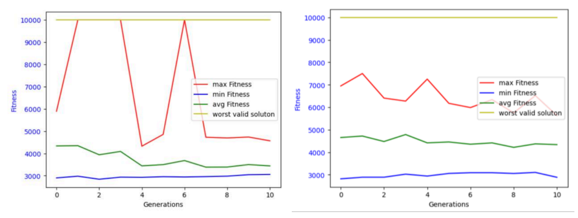
After the first experiment, we used new approaches to gain better results in the actual map and the fitness factor. You will see all of these approaches below.

A. Repair Function

One of the big problems during the first experiment was that we constantly generate individuals during the iterations. Nevertheless, we created a repair function to prevent these problems. This function has the duty to prevent to generate duplicated genes and the genes that put a camera on walls.

B. Map Boundaries

After some experimentation, we found out that when the cameras are near the edge of the map their vision will be considerably decreased. So, we calculated some boundaries over the whole map like Fig. 3. We even prevent the random functions to generate any numbers out of this boundary.



(a) Without repair function *(b) With repair function*

Fig. 2

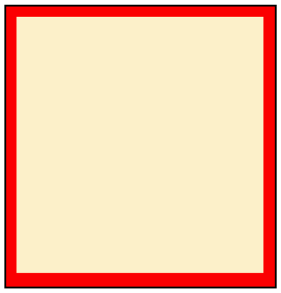


Fig. 3: Red parts are the boundaries line, and the yellow parts are the valid part of the map.

C. *New Wall Variable*

Cameras near the wall act exactly like when they are at the edge of the map. In a result, we tried to create a new walls variable to push the cameras from near the wall to the center of the map for more vision. The new wall variable has the same walls with occupied cells near them, which the cameras can't be there too. We add the previous Map Boundaries to this variable too, it will help to prevent putting cameras in these places even during the iterations. You can see the examples in Fig. 4.

D. *Distance Between Cameras*

Cameras near each other can overlap their visions. This phenomenon can decrease their scope. So, we wrote a function to calculate their Euclidean distance. In the result, in initialization function we will check that two cameras are not initiated near each other. This helped us to prevent useless individuals.

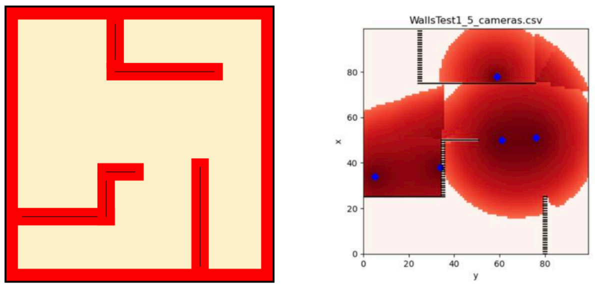


Fig. 4: Black lines are the walls with their boundaries (red outlines). The right map shows a sample of solution without this approach.

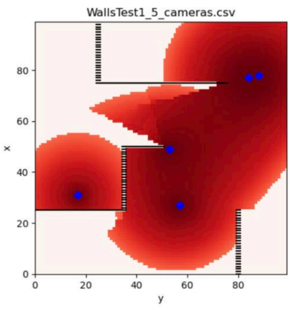


Fig. 5: Sample solution without using distance between cameras approach.

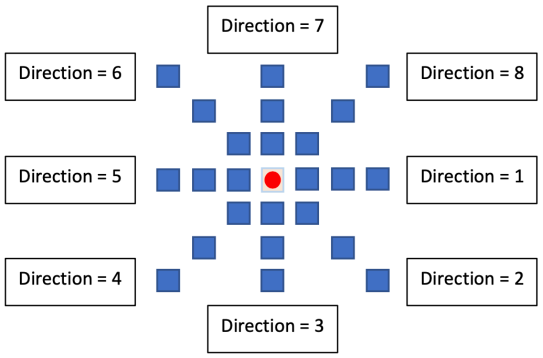


Fig. 6: The red circle is the current cell. The blue squares are the possible movements in 8 different directions.

E. *Custom Mutation (move to your neighbor cells)*

For enhancing the diversity, we needed to create a custom mutation to find better solutions. The custom mutation will generate two random numbers called epsilon and direction. Epsilon is the number of steps the cell should take. Direction is the direction that the cell should go. For example, if the epsilon is 2 and the direction is 1 the cell should go two steps to the right. You see all possible movements in Fig. 6. If the new movement is not valid (out of the map or not in the boundaries or on the wall) we will generate a new random gene in the valid boundaries.

V. New Results

The approaches mentioned helped us to place these cameras in better cells. The iterations become longer. And we experimented with more generations and populations. You can see the results in Figure 7. These results are without random numbers seeding. Because after you seed random numbers between 0 to 10000 you will get 99 unique numbers which harm our diversity.

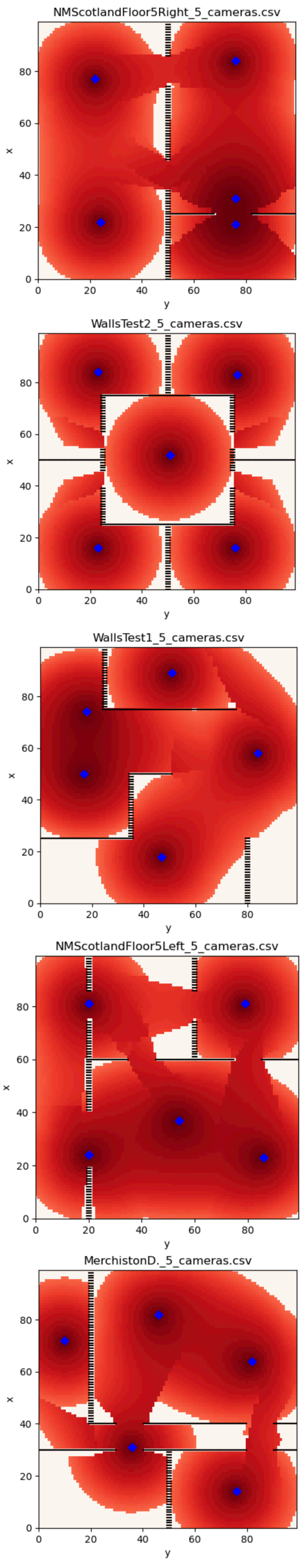


Fig. 7: Visualizations

Iter / params	Fitness	Time (s)	Camera Coordinates
(10, 100)	964.0	13.462	(17, 78)(66, 71)(77, 87)(22, 19)(77, 25)
(10, 150)	745.0	19.598	(74, 24)(49, 82)(11, 70)(80, 73)(26, 19)
(15, 200)	802.0	37.379	(48, 28)(21, 79)(77, 78)(77, 24)(13, 23)
(15, 250)	892.0	46.304	(76, 23)(81, 91)(22, 21)(19, 78)(63, 62)
(30, 250)*	746.0	90.024	(77, 24)(17, 22)(81, 80)(49, 36)(21, 81)
(40, 400)	772.0	188.504	(22, 77)(24, 22)(76, 21)(76, 31)(76, 84)

TABLE II: Results for the NMScotlandFloor5Right_5_cameras, Average Fitness= 820.17

Iter / params	Fitness	Time (s)	Camera Coordinates
(10, 100)	2169.0	13.122	(60, 52)(16, 24)(16, 76)(77, 86)(78, 17)
(10, 150)	2057.0	19.899	(23, 83)(78, 82)(51, 53)(79, 17)(21, 22)
(15, 200)	2012.0	37.865	(23, 83)(23, 18)(77, 84)(51, 52)(76, 16)
(15, 250)	2040.0	46.949	(76, 18)(21, 17)(50, 52), (77, 83)(22, 82)
(30, 250)	2009.0	89.937	(23, 84)(51, 50)(23, 16), (77, 17)(77, 83)
(40, 400)*	2008.0	193.093	(23, 84)(23, 16)(51, 52)(77, 83)(76, 16)

TABLE III: Results for the WallsTest2_5_cameras, Average Fitness= 2049.17

Iter / params	Fitness	Time (s)	Camera Coordinates
(10, 100)	2041.0	13.329	(89, 68)(18, 73)(14, 50)(53, 20)(49, 84)
(10, 150)	2020.0	20.555	(49, 88)(54, 19)(88, 68)(14, 49)(20, 74)
(15, 200)	1996.0	38.627	(24, 72)(39, 17)(16, 51)(83, 48)(60, 89)
(15, 250)	1955.0	48.039	(23, 74)(43, 17)(13, 49)(52, 88)(85, 54)
(30, 250)*	1687.0	91.126	(19, 58)(83, 32)(49, 87)(33, 14)(85, 73)
(40, 400)	1936.0	189.378	(17, 50)(84, 58)(51, 89)(18, 74)(47, 18)

TABLE IV: Results for the WallsTest1_5_cameras, Average Fitness= 1939.17

Iter / params	Fitness	Time (s)	Camera Coordinates
(10, 100)	1020.0	13.921	(67, 78)(20, 77)(78, 58)(79, 13)(23, 26)
(10, 150)	1308.0	20.132	(24, 25)(19, 77)(77, 80)(77, 22)(40, 80)
(15, 200)	1249.0	41.408	(41, 74)(19, 80)(80, 80)(20, 24)(76, 25)
(15, 250)	968.0	49.783	(76, 13)(20, 25)(80, 80)(20, 80)(68, 52)
(30, 250)*	612.0	91.006	(20, 81)(19, 24)(76, 58)(60, 79)(76, 8)
(40, 400)	893.0	194.215	(20, 24)(79, 81)(20, 81)(54, 37)(86, 23)

TABLE V: Results for the NMScotlandFloor5Left_5_cameras, Average Fitness= 1008.33

Iter / params	Fitness	Time (s)	Camera Coordinates
(10, 100)	2452.0	13.060	(74, 8)(86, 41)(9, 71)(35, 33)(55, 86)
(10, 150)	2364.0	20.705	(80, 22)(22, 72)(22, 16)(9, 54)(76, 77)
(15, 200)	2001.0	37.921	(11, 72)(35, 31)(80, 71)(48, 75)(76, 14)
(15, 250)	2000.0	47.244	(47, 81)(80, 68)(76, 18)(36, 31)(14, 71)
(30, 250)	2115.0	91.695	(8, 55)(82, 77)(48, 66)(76, 16)(22, 16)
(40, 400)*	1950.0	192.301	(82, 64)(76, 14)(36, 31)(10, 72)(46, 82)

TABLE VI: Results for the MerchistonD._5_cameras, Average Fitness= 2147.0