Path Finding

Description: This repository contains the short-project #2 and #3 of Artificial Intelligence course from Instituto Tecnológico of Costa Rica, imparted by the professor Juan Manuel Esquivel. The project consists on implementing search algorithms to solve labyrinth problems. Specifically, we will solve a problem where a board introduced as a file have a rabbit and carrots. The rabbit, then will have to find a specified amount of carrots.

The project will be divided into two parts, both contained in a console program that will execute with different modes using differentiated flags. The first will consist in developing a heuristic used within A* to generically go over the board looking for carrots. The second will consist of developing a Genetic Algorithm that will optimize the placement of directional signals so that the rabbit travel over the board.

Content:

Installation:

Before using the project, first you have to install all the project dependencies.

- Python 3.4 or greater.

Now that all dependencies are installed. You can install the project using:

`pip install -U tec.ic.ia.pc2.g07.main`

Or you can clone the repository by:

`git clone https://github.com/mamemo/Path-finding.git`

Usage:

When you have the directory with the repository, you have to search that repository on a console and execute the instruction:

`python -m tec.ic.ia.pc2.g07.main -h`

This will display all the flags that you can work with:

```
  -h, --help            show this help message and exit
  --a-estrella          A* Algorithm.
  --vision VISION       Vision field range.
  --zanahorias ZANAHORIAS
                        Objective's amount to search.
  --movimientos-pasados MOVIMIENTOS_PASADOS
                        Number of past movements to store. (Default 5)
  --genetico            Genetic Algorithm.
  --derecha             All individual going to the right.
  --izquierda           All individual going to the left.
  --arriba              All individual going up.
  --abajo               All individual going down.
  --individuos INDIVIDUOS
                        Individual's amount.
  --generaciones GENERACIONES
                        Generation's amount.
  --politica-cruce {random,sons_of_sons}
                        Crossover algorithm.
  --cantidad-padres CANTIDAD_PADRES
                        Number of parents.
  --tasa-mutacion TASA_MUTACION
                        Mutation Rate Percent.
  --tablero-inicial TABLERO_INICIAL
                        Input file destination. File containing the scenario
                        to be resolved.
```

Each algorithm uses differentiated flags, but they have one in common, you can see the description next to the flag:

```
  --tablero-inicial TABLERO_INICIAL
                      Input file destination. File containing the scenario
```

```
                         to be resolved.
```

To run A* you will need:

```
  --a-estrella          A* Algorithm.
  --vision VISION       Vision field range.
  --zanahorias ZANAHORIAS
                        Objective's amount to search.
  --movimientos-pasados MOVIMIENTOS_PASADOS
                        Number of past movements to store. (Default 5)
```

To run Genetic Algorithm you will need:

```
  --genetico            Genetic Algorithm.
  --derecha             All individual going to the right.
  --izquierda           All individual going to the left.
  --arriba              All individual going up.
  --abajo               All individual going down.
  --individuos INDIVIDUOS
                        Individual's amount.
  --generaciones GENERACIONES
                        Generation's amount.
  --politica-cruce {random,sons_of_sons}
                        Crossover algorithm.
  --cantidad-padres CANTIDAD_PADRES
                        Number of parents.
  --tasa-mutacion TASA_MUTACION
                        Mutation Rate Percent.
```

Board Representation:

The board will be a text file of N lines and M columns and each character can be only:

- C: capitalized, identifies the position of the rabbit. There can only be one per board.
- Z: capitalized, identifies the position of a carrot. There may be multiple carrots per board.
- Blank space: The space character does not have any side effects but it does must be present to indicate a position on the board by which the rabbit can transit.
- <: symbol for smaller than, indicates a change of direction to the left.
- >: symbol for greater than, indicates a change of direction to the right.
- A: letter A capitalized, indicates a change of direction upwards.
- V: letter V capitalized, indicates a change of direction downwards.
- Change of line: It has no interpretation in the program other than separating different rows.

Examples of boards will be presented at the analysis of the algorithms.

Algorithms' Report

This section contains the analysis of using each algorithm and how well it performs with different parameters. Every model is called from main.py and the process is the following:

1. Waiting parameters.
2. Receiving parameters.
3. Creating the respectively algorithm.
4. Validate the flags.
5. Running the algorithm.
6. Generate output messages and files.

A*

The main goal was to create a program that moves the rabbit through the board, a box at a time, using A*. We had to design a cost function to guide the search A* considering the accumulated cost and a heuristic to approximate the future cost. The accumulated cost up to a specific point in the execution of the algorithm will be simply the number of steps that the rabbit has given.

The future cost will be describe below, the elements that we had to consider was:

- The environment is not completely observable.
- The rabbit will have a range of vision. If the current state has the rabbit in the box (20, 18), for example, and the rabbit has a vision of two, may take into account the content of the boxes (18, 16) to the box (22, 20).

- The design of the heuristic should not include "memory" that makes the environment implicitly observable. In particular, the algorithm should maintain the memory footprint equally for "big" and "little" search spaces.
- At the beginning of the program, it will be indicated how many carrots the rabbit should look for in total to finish his work.

The function cost is defined as follows:

$$f(x) = g(x) + h(x)$$

we defined g(x) as:

$$g(x) = \begin{cases} g(x)+1 & if x \neq 0 \\ 1 & if x = 0 \end{cases}$$

where x stands for the number of steps taken.

Then we have the heuristic h(x) defined as:

$$h(x) = h_{without\_carrots(x)} + h_{with\_carrots}(x)$$

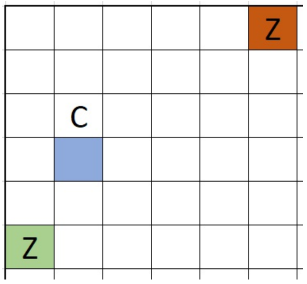
We defined h_without_carrots(x) as the function of heuristic without considering the carrots in sight:

$$h_{without\_carrots(x)} = \begin{cases} 10 & if\ x\ is\ last\ state \\ 5 & if\ x\ is\ in\ last\ movements \\ 1 & if\ x\ is\ not\ passed \end{cases}$$

where x is the possible state to move. Then the function considering the carrots is defined as the heuristic h_with_carrots(x):

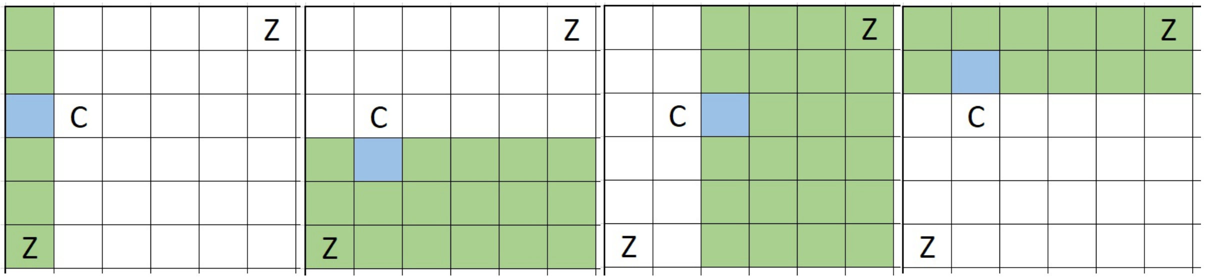
$$h_{with\_carrots(x)} = nearest\_carrot + carrots\_at\_reach$$

The *nearest_carrot* part is the distance between the possible state and the nearest possible carrot. For example in:



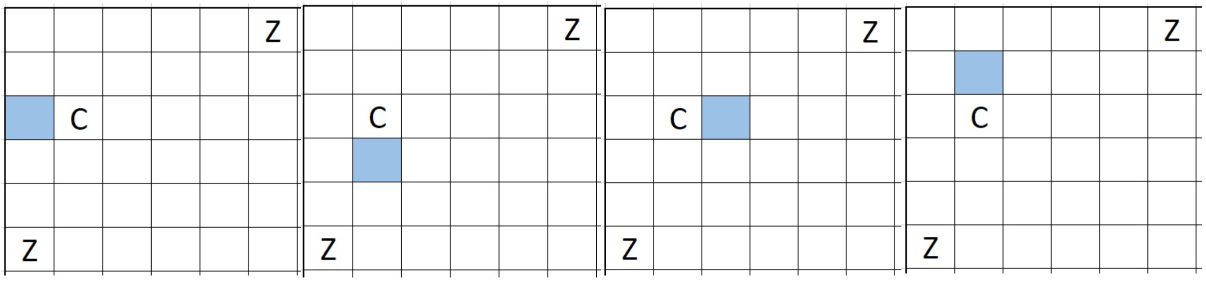
For the tested next state **[3,1]**, the nearest carrot will be **[5,0]**; because abs(3-5)+abs(1-0) = **3** and for the other carrot in sight the distance is abs(3-0)+abs(1-5) = **7**.

The *carrots_at_reach* part is how many carrots are at reach if you move to the state. For example, the reach fields are presented in the next images:



where the results for the states [[2,0], [3,1], [2,2], [1,1]] would be **1**.

So, as summary, if we have these scenarios, the costs are:

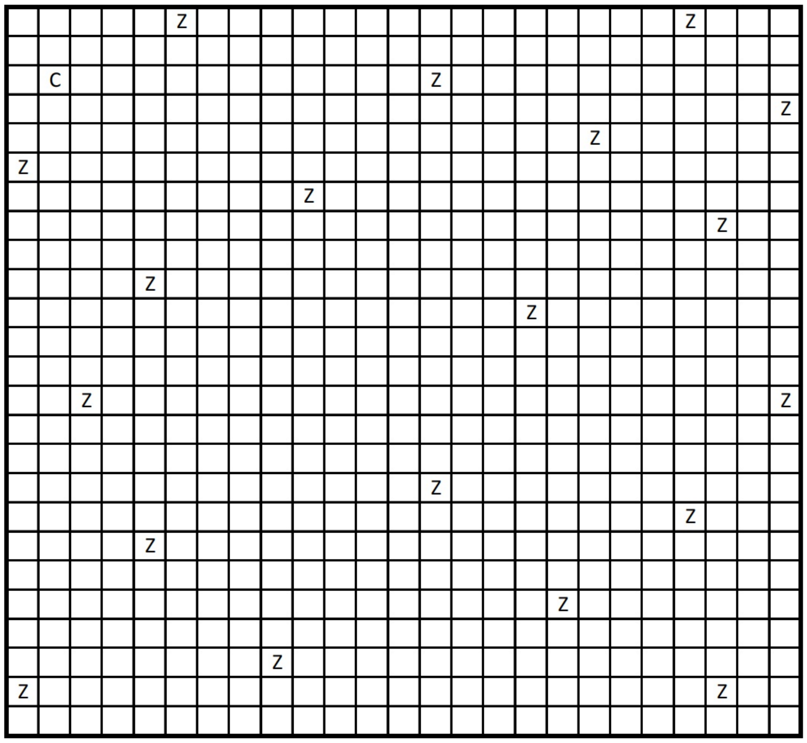


$$\begin{array}{cccc}
g(x) = 1 & g(x) = 1 & g(x) = 1 & g(x) = 1 \\
h_{without\_carrots} = 1 & h_{without\_carrots} = 1 & h_{without\_carrots} = 1 & h_{without\_carrots} = 1 \\
h_{with\_carrots} = 3 + 1 & h_{with\_carrots} = 3 + 1 & h_{with\_carrots} = 5 + 1 & h_{with\_carrots} = 5 + 1 \\
h(x) = 5 & h(x) = 5 & h(x) = 7 & h(x) = 7 \\
f(x) = 6 & f(x) = 6 & f(x) = 8 & f(x) = 8
\end{array}$$

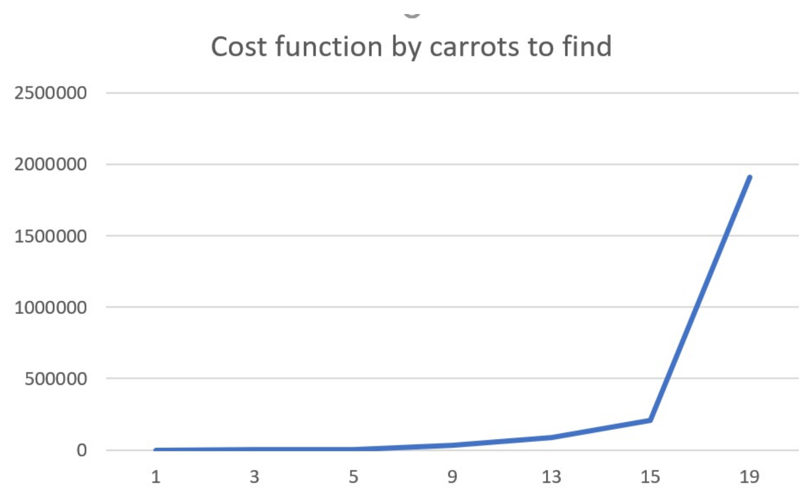
The best possible states are equally [2,0] and [3,1] with a cost of **6**. In these cases the next state is selected randomly.

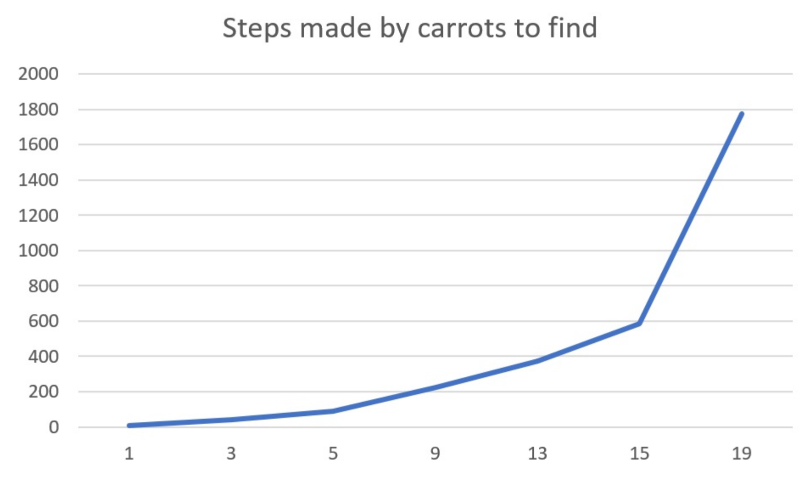
Now, we will present the analysis of cost variation when carrots number and vision are changed. Every each of the following experiments are presented after 10 executions. The value below is the mean of the cost. The commented code of this algorithm is in A_Star.py.

The used board is presented below and it contains 25x25 boxes and 19 carrots. You can find it on test_board.txt.



Variation in carrots number





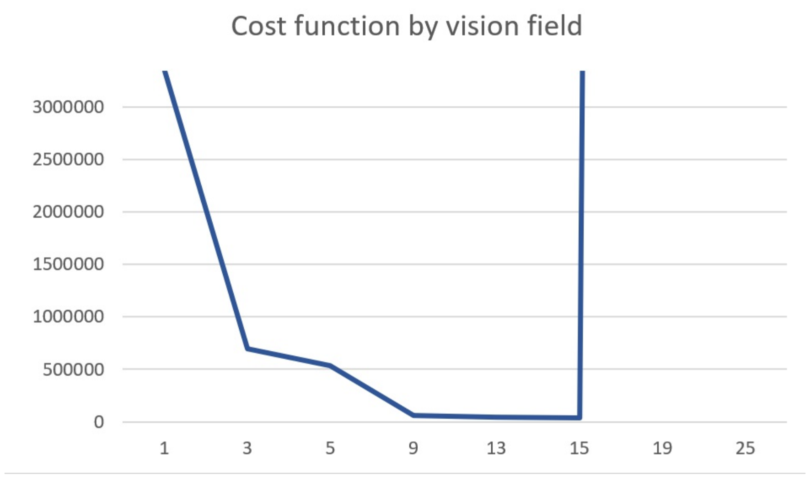
> The vision field is 2.

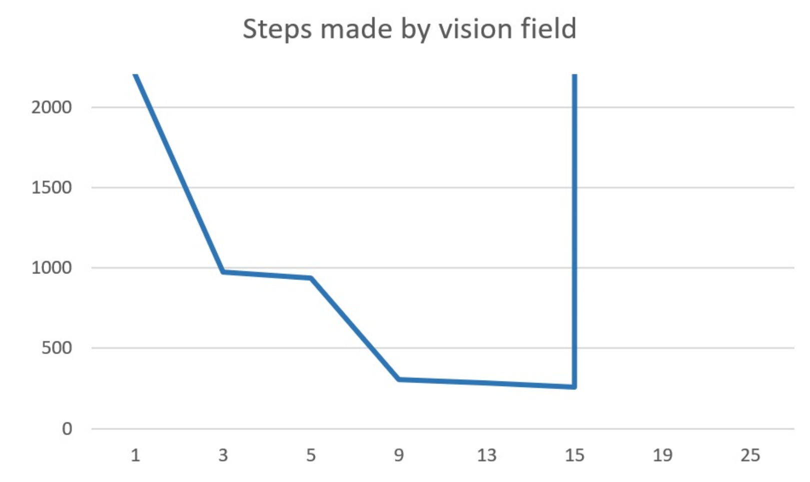
Carrots	1	3	5	9	13	15	19
Cost	25.3	929.5	4707.5	30697.7	85406.3	210162.4	**1911582.1**
Steps	6.6	41.6	90.8	225.7	375.3	587.5	**1776**

From these results we can conclude:

- If the rabbit needs to find more carrots will have to move more around the board and the cost will grow.

Variation in vision field





> The carrots to find are all of them. (19)

Carrots	1	3	5	9	13	15	19	25
Cost	3343809.3	699767.4	531807.1	56611.2	44365.3	**36510.9**	Inf	Inf
Steps	2203.2	973	935	304.4	282.6	**257.4**	Inf	Inf

From these results we can conclude:

- On vision field of 19 and 20, the rabbit gets in a loop so it will never stop, that's why in the chart we put it a Inf value.
- The loop is caused because the rabbit walk to a certain point where it approaches a carrot and another carrot see as a better approach and viceversa. So it moves in circles.
- As we can see when the vision field gets bigger the rabbit can do better decision, so it reduces the cost and steps. That tendency remains until a certain number of vision field.

Genetic Algorithm

The main goal was to create a program that evolve the board with directional signals. The algorithm should not make decisions about how to move the rabbit step by step, but assume that the rabbit always moves in one way and it can change its direction with signals. The program, as said before, will have to determine the ideal location of the signals so that the rabbit can collect ALL the carrots on the board in as few steps as possible and with the least amount of signals.

The initial state is given by a file equal to A*. In addition, it should be indicated through a flag, what will be the initial direction of the rabbit, which is key to place the first signal. Each individual, for the genetic algorithm, will be a complete board with its corresponding carrots, signals and location of the rabbit.

We had to follow certain implementation restrictions:

- Mutations are punctual operations. Add a signal in a random box, change the direction of a signal or remove a signal.
- The fitness function must combine total carrots collected, amount of rabbit steps and number of signals. It must be emphasized that it is valid to have a linear combination heuristic where it is reflected that there is a greater penalty for adding a signal than for giving a certain number of steps.
- The fitness function does not influence the selection of crossover and mutation. Both basic operations of genetic algorithms are, in essence, completely random given a policy. The fitness function only serves to order the population of more to less fitness and select those that pass to the next generation.

Next we will explain the mutation, crossover and fitness decisions and implementation:

- Mutation: We had a mutation rate given by parameter, if a random generated number is below the mutation rate, that individual will be mutated. If it qualify to be mutated, a random box is selected, if that random box contains "*C*" or "*Z*" (rabbit or carrot) that mutation will be discarted. Otherwise, a random element is selected from these options "*>*", "*<*", "*V*", "*A*", " ", the element that is already on the box is removed from the options. The commented code can be found on Genetic.py at *mutate_population()*

> The original individual is NOT changed, a mutated copy is added in the population instead.

- Crossover: The crossover algorithm is indicated by parameter. We had to implement 2 kind of crossover, we implemented Random and Son of sons crossover. Basically both of them has 2 functions:
    - *select_parents(population)*: This function selects a number (given by parameter) of parents from the population. For both implemented crossovers, the parents are selected randomly.
    - *cross(parents)*: According to each crossover this function works different. Its objective is to cross the selected parents. **Random** crossover divide the parents in segments (same as number of parents) and each parent give the respectively segment according to the position of its selection, finally the son is added to the population. For example: If selected parents are [[1,2,3,4,5,6], [7,8,9,10,11,12]], the child would be [1,2,3,10,11,12]. And it can scalate to *n* parents. **Son of sons** simulates a more "*realistic*" idea of a generation. Taken for example the human life, a typical individual live to create a son and that son create another child. So, that is basically what this crossover do, the selected parentes are divide in two groups, each group creates a son like Random crossover does and then those sons creates a grandson that will be added to the population. The grandson's parents are randomly sorted and are crossed like Random crossover does.
- Fitness: To give each individual a fitness number, *walk_trough_board* and *calculate_fitness* functions were implemented in Genetic.py. In order to calculate the fitness we need 4 values given by *walk_trough_board*, those are:
    - Movements: As it's said, is the number of steps made by the rabbit. It can be from 0 to the amount of boxes.

    > The *walk_trough_board* is stopped when movements are the same as number of boxes in the board because that means a loop.

    - Carrots Eaten: Is the number of carrots that the rabbit could find. If the carrot has already been found this number don't increment. It can be from 0 to the total number of carrots.
    - Useful signals: The amount of signals that lead to a carrot. Signals that don't lead to a carrot or weren't used don't increment this number. It can be from 0 to

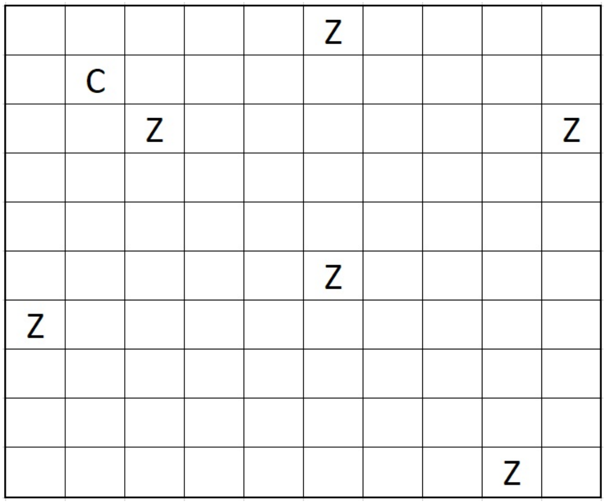
number of boxes minus the total amount of carrots minus one (the rabbit).
- Has opposite signals: A flag indicating if the board has opossite signals. It can be True or False. For example: [">"," ","<"] woud result as True.

After we got those values, we give penalties and rewards to the individual, those are:

Case	Points	Explanation
Carrot eaten	+100	Reward for find a carrot. Applied for each carrot found.
Movement	+5	Reward for step. Applied for each step. Indicating that the individual should explore more.
Useful Signal	+3	Reward for having useful signals. Applied for each useful signal.
Carrot left	-50	Penalty for not find a carrot. Applied for each carrot left.
Useless signals	-50	Penalty for having useless signals (Don't lead to a carrot or weren't used). Applied for each useless signal.
Starting	-50	Penalty for not having signals and don't get all the carrots. Applied once. Works when the algorithm is starting, giving better results to individuals that do something.
Has opposite signals	-1000	Penalty for having opposite signals. Applied once.
Trap in a loop	-1000	Penalty for geting the movements same as the number of boxes in the board. Applied once. Discard the individual completely.

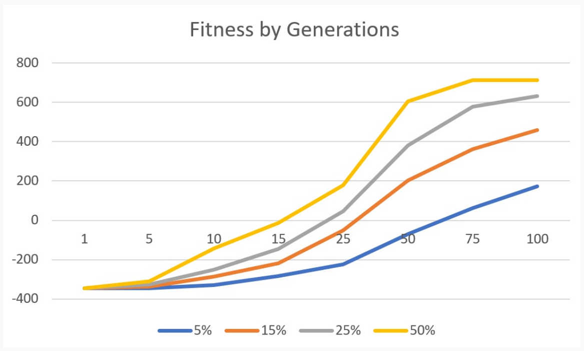
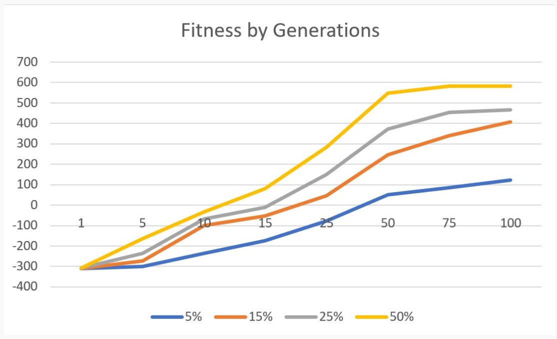
Now, we will present the analysis of fitness variation when mutation rate and crossover are changed. Every each of the following experiments are presented after 30 executions. The value below is the mean of the fitness. The commented code of this algorithm is in Genetic.py.

The used board is presented below and it contains 10x10 boxes and 6 carrots. You can find it on board_example.txt.

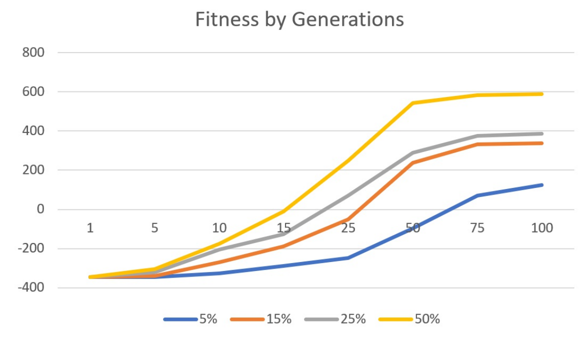


					Z				
	C								
		Z							Z
					Z				
Z									
								Z	

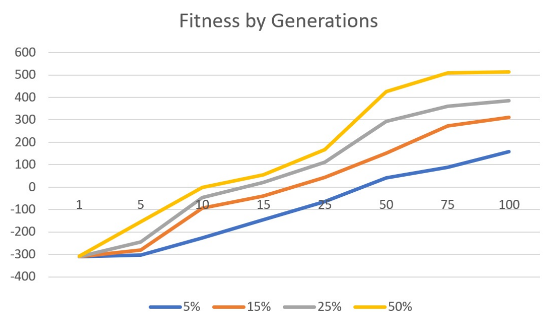
Mutation Rate Variation

Left		**Right**	
			

Up



Down



The number of individuals are 100.

The random crossover is selected.

The number of parents is 2.

Direction	**Left**								
Generation		**1**	**5**	**10**	**15**	**25**	**50**	**75**	**100**
Mutation Rate	**5%**	-345	-344.34	-327.57	-283.37	-223.78	-70.82	61.08	173.93
	15%	-344.91	-336.81	-285.48	-49.88	-217.04	202.808	360.49	458.01
	25%	-344.67	-325.77	-249.44	-145.51	47.209	379.55	577.84	632.06
	50%	-344.58	-310.003	-143.508	-12.36	177.43	605.52	711.93	**711.93**

Direction	**Right**								
Generation		**1**	**5**	**10**	**15**	**25**	**50**	**75**	**100**
Mutation Rate	**5%**	-309.72	-300.31	-235.91	-174.97	-77.63	50.55	85.08	122.6
	15%	-308.95	-272.13	-100.29	-53.37	47.003	247.06	340.46	407.512
	25%	-308.63	-235.71	-67.27	-11.48	149.78	371.77	453.6	466.8
	50%	-307.41	-165.005	-32.78	79.47	282.86	548.65	581.53	**581.53**

Direction	**Up**								
Generation		**1**	**5**	**10**	**15**	**25**	**50**	**75**	**100**
Mutation Rate	**5%**	-344.91	-344.12	-325.37	-287.501	-247.25	-95.85	69.17	123.91
	15%	-344.75	-340.26	-269.709	-188.32	-50.72	236.406	331.72	338.37
	25%	-344.505	-319.506	-206.11	-127.73	69.65	290.21	375.43	387.3
	50%	-344.34	-304.73	-174.45	-11.29	247.91	541.63	583.106	**588.73**

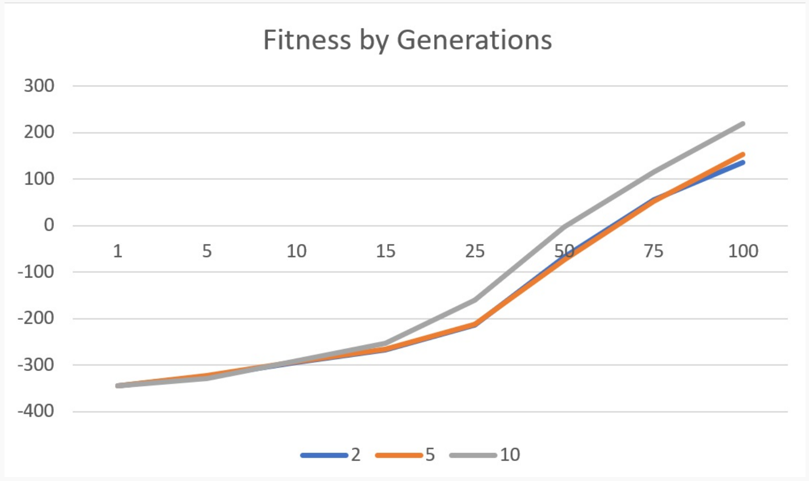
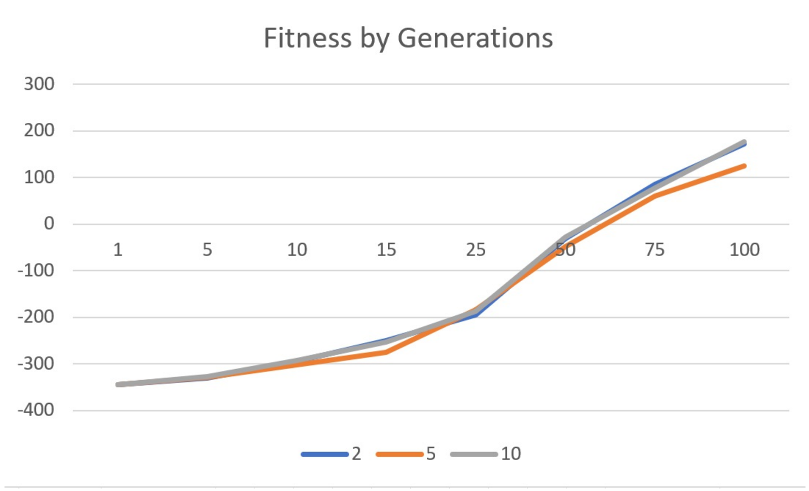
Direction	Down

Generation		1	5	10	15	25	50	75	100
Mutation Rate	**5%**	-309.801	-301.49	-225.82	-145.09	-64.05	41.35	88.12	159.29
	15%	-309.29	-280.55	-94.39	-39.86	44.33	152.58	273.103	311.9
	25%	-308.84	-245.0003	-46.47	21.093	111.85	293.44	361.28	385.83
	50%	-307.73	-153.14	-1.006	54.705	166.708	424.99	508.16	**514.63**

From these results we can conclude:

- As the mutation rate gets higher, also does the average fitness. This tendency goes for every direction.
- Higher mutation rate means more modified individuals and the probability of mutate to a better individual gets higher.

Crossover Variation

Random	
Son of Sons	

> The number of individuals are 10. A small number of individual is selected to emphasize more on crossover than mutation.
>
> The direction is left because is one of the hardest direction using the specified board.
>
> The mutation rate is 50%.

Crossover	Random

Generation		1	5	10	15	25	50	75	100
Parents	**2**	-343.77	-324.16	-294.3	-267.43	-214.48	-66.77	55.47	135.001
	5	-344.018	-323.073	-292.08	-264.93	-211.84	-73.76	51.46	152.49
	10	-344.26	-328.76	-290.95	-252.49	-160.93	-2.95	115.93	**219.77**

Crossover	Son of Sons

Generation		1	5	10	15	25	50	75	100
Parents	**2**	-344.99	-329.86	-293.93	-250.46	-194.88	-31.87	85.12	171.18
	5	-344.25	-328.62	-302.38	-275.8	-184.56	-47.95	59.931	124.26
	10	-344.24	-326.37	-292.24	-253.37	-186.69	-27.84	77.43	**176.805**

From these results we can conclude:

- For both crossovers, the best results was given on the maximum amount of parents. That's because all the population gives information to create a new individual.
- However, this configuration (maximum amount of parents) can't be use in real solutions that need hundreds of individuals.
- Both crossovers did it pretty the same but Random did it better at the end. That's because Random crossover leaves more "*clean*" the passed information, meanwhile on Sons of sons crossover there can be details that the sons don't pass to the grandson.
- However, Son of sons can produce the same results using the two parents or the total of them.

Unit Testing:

For this project we created unit testing for every function that had certain relevance on the algorithm. All the tests can be located at unit_tests were every test is documented with an objective, tested function and desired outputs.