Artificial Intelligence Fundamentals Report

Giacomo Aru, Gabriele Benanti, Giulia Ghisolfi, Luca Marini, Irene Testa
Master Degree in Computer Science, University of Pisa, Academic Year 2022-2023
January 27, 2023

Abstract

The aim of this work is to build an artificial agent able to play the game Snake. In this report we present two different algorithms to implement the agent program and we analyze and compare their performances.

1 Introduction

Snake is a well-known video game in which a snake moves through a grid and grows as it collects apples. Designing an agent able to play the game is a challenging task that provides an ideal context for practicing Artificial Intelligence algorithms. Indeed, in 2018 it has also drawn the attention of the non-profit research organization OpenAI [5]. In this work we tackled the problem using methodologies and techniques seen during the Artificial Intelligence Fundamentals course. To design and develop the agent algorithms we integrated our solutions with the ones proposed in similar projects (e.g. [4], [6], [7]).

2 Methodologies

The game has been developed using the Python programming language and the **pygame** library. With this library we implemented also a graphical interface that allows the user to choose the algorithm to run and watch the agent play. The implementations of the search algorithms we used were taken from the GitHub repository of the book *Artificial Intelligence a Modern Approach* [3].

Before presenting the implementation of the agent program, we first provide a description of the game environment and of the measures we used to evaluate the agent performance.

The game map is a $n \times m$ grid that may contain obstacles and the game starts with the snake placed in the top-left corner of the grid, facing south and occupying three cells of the grid. At each game step, the snake can move forward, turn left by 90°, or turn right by 90°. When the snake's head overlaps the apple, a new apple is placed randomly on a free cell of the grid and at the successive game step the body of the snake lengthens by one unit. The game ends when the snake collides with its tail, with the frame of the grid or with the obstacles in it, and it is won when the snake covers the whole grid with its body, i.e. when its length is equal to the grid area. Generally, in the video game, if the player does not specify a move within a certain time interval, the snake keeps moving in the same direction it was moving at the previous step. In the design phase of the agent we decided to relax this time constraint, allowing an agent to have, in principle, an infinite amount of time to decide the next move. However, to reflect the traditional game settings, we tried to limit the time complexity of the operations needed to compute the agent moves.

The objective of the agent is to collect as many apples as possible in as few game steps as possible. Therefore, to evaluate the performance of the agent we measured the length of the snake at the end of the game and the number of moves the snake made.

To implement the agent program we developed two different algorithms. The first uses a greedy approach which mainly evaluates the direct consequences of the agent actions. The second, instead, is more long-term oriented and relies on a Hamiltonian cycle, i.e. a circuit that passes

through each cell of the game grid exactly once. The logic behind these algorithms is described in the next two subsections.

2.1 Greedy Algorithm

The problem of finding a path to the apple can be formulated as a search problem whose state space can be represented as a graph: the vertices of the graph correspond to the free cells in the grid and the edges correspond to the available actions. Following the shortest path to the apple, without considering the position of the snake's tail, can easily lead the snake to trap itself. To overcome this problem we developed a strategy that searches for "safe paths". It works as follows: (1) when an new apple appears, compute a path between the snake's head and the apple; if it exists then (2) compute the position of the snake once it has reached the apple through that path, and (3) if there exists a path between the snake's future head and its future tail, make the snake follow the path computed at (1); else "extend" the safe path previously computed[1] trying to make it pass through as many grid cells as possible, hoping to reach the apple.

To compute the path required at step (1) we used weighted A* and two heuristic functions derived from the Manhattan distance. One of the heuristics is used to avoid the snake taking zigzagging paths. The other heuristic function aims at "compressing" the snake's body to make it exploit better the free space on the grid. The heuristic to use and the weights of A* can be chosen as parameters of the algorithm.

To "extend" the safe path when the condition checked at step (3) is not satisfied we used a heuristic algorithm that computes an approximation of the longest path between two nodes on a grid.

With the just described algorithm the snake could get stuck in a loop, especially when it is long and has no other possibilities rather than eating the apple and consequently die. To avoid this problem, when a loop is detected the snake abandons the safe path and moves toward the apple even if it will die, so that the performance measure can be maximized.

2.2 Hamiltonian Algorithm

An always winning strategy to play the game Snake consists in following an Hamiltonian cycle: sooner or later the snake will eat each apple and it will never crash against itself. However, following an Hamiltonian cycle requires on average to traverse half of the free grid cells to reach each apple. To reduce the number of steps needed to complete the game we developed two different strategies: the *Hamiltonian Shortcuts Strategy* and the *Hamiltonian Repair Strategy*. Both strategies rely on a fixed, precomputed Hamiltonian cycle, therefore they cannot be used on odd sized grids (grids with odd width and odd height) because on those grids an Hamiltonian cycle does not exist (for a proof see e.g. [1]).

The idea behind the *Hamiltonian Shortcut Strategy*, firstly proposed in [7], is to skip sections of the Hamiltonian cycle to reach apples faster while avoiding moves which lead to crashes. The key insight to implement this strategy is that by unrolling the Hamiltonian cycle on a line, we only need to check, for each possible move, if the snake's body "remains ordered" on the cycle (see Figure 2 in Appendix 4). Checking if this condition is satisfied requires to compute the relative position on the cycle of the apple and of the snake's current and future head with respect to the snake's tail, which can be done in constant time. Unfortunately, taking shortcuts in the later stages of the game can occasionally lead the snake to crash into itself. To avoid such early deaths, the snake stops taking shortcuts once it has achieved a certain length, which can be configured as a parameter of the algorithm. Beyond that length the snake simply follows the Hamiltonian cycle.

[1] Here we are assuming that at the start of the game a safe path always exists.

With the *Hamiltonian Repair Strategy*, instead, the agent tries to repair the Hamiltonian cycle it is following by building a new Hamiltonian cycle in which the distance between its head and the apple is smaller. The time complexity of this operation is linear in the number of cells of the game grid, but since in the later stages of the game it does not shortens the length of the path to the apple significantly, we decided to apply it only when the snake's body length is smaller then a certain value, which can also be configured as a parameter of the algorithm.

Both the strategies described above can be used to compute the next move: the agent first tries to repair the cycle it is following, and then looks for shortcuts to further reduce the number of steps required to reach the apple.

3 Results

To evaluate the performances of the algorithms we tested them with different parameter configurations (see Tables 2 and 5 in the Appendix), reporting the time needed to compute each move and the length of the snake at each step of the game. Tests were conducted using different machines, therefore the time measured is subjected to the performances of the processors used (as evidenced by Figure 5 in the Appendix).

In all the 100 games run on 10×10 grids the agent with the Hamiltonian Algorithm always managed to win and the lowest number of moves required to win (on average), with one of the tested configurations, was around 1890. With the same test settings the highest percentage of games won by one of the configurations of the Greedy Algorithm was 32.9%, requiring on average 1600 moves. We noted that the percentage of games won with the Greedy Algorithm decreases on wider grids (with a particular parameter combination the agent won 62% of the times on a 6×6 grids, 30% on a 10×10 grid and 12% on a 16×16 grid). With almost all the tested Greedy configurations the average length achieved by the snake at the end of the game was 98% of the grid size.

The maximum time needed to compute each move on both 10×10 and 16×16 grids over 100 runs was around 20 milliseconds with the Greedy Algorithm and around 8 milliseconds with the Hamiltonian Algorithm. Although, on 10×10 grids both algorithms require, most of the time, around 5 milliseconds to compute the next move (human time reaction to visual stimuli is around 200 milliseconds [2]). We observed that with the Greedy Algorithm the number of moves required to grow the snake are less than the ones needed by the Hamiltonian Algorithm through almost all the game except when the snake has covered most of the grid, in these last few stages of the game the Hamiltonian Algorithm needs less moves to reach the apples.

Further details on the results obtained testing the agents are reported in Tables and Figures in the Appendix.

4 Conclusion

In this work we implemented and tested two agent programs to play the game Snake. One uses a greedy approach (Greedy Algorithm), the other a more long-term oriented approach (Hamiltonian Algorithm). The Greedy Algorithm can be used on any type of grids, with any type of obstacle configuration; the Hamiltonian Algorithm, instead, requires a pre-computed Hamiltonian Cycle (not all grids may have one). With the Hamiltonian Algorithm the agent won in all the tests made, with the Greedy Algorithm it won around 30% of the times on 10×10 grids and when it didn't it covered 98% of the grid. The Greedy Algorithm outperforms the Hamiltonian Algorithm in terms of the number of moves required to grow the snake in the first stages of the game.

References

[1] S. D. Chen, H. Shen, and R. Topor, "An efficient algorithm for constructing hamiltonian paths in meshes", *Parallel Computing*, vol. 28, no. 9, pp. 1293–1305, 2002.

[2] A. Jain, R. Bansal, A. Kumar, and K. Singh, "A comparative study of visual and auditory reaction times on the basis of gender and physical activity levels of medical first year students", *International Journal of Applied and Basic Medical Research*, vol. 5, no. 2, p. 124, 2015.

[3] S. J. Russell and P. Norvig, *Artificial intelligence a modern approach*. Pearson Education, 2022.

[4] AI for the game Snake. `https://github.com/chuyangliu/snake/blob/master/docs/algorithms.md`.

[5] OpenAi, Requests for Research 2.0. `https://openai.com/blog/requests-for-research-2/`.

[6] How to Win Snake: The UNKILLABLE Snake AI. `https://github.com/BrianHaidet/AlphaPhoenix/tree/master/Snake_AI_(2020a)_DHCR_with_strategy`.

[7] Nokia 6110 Part 3 – Algorithms. `https://johnflux.com/2015/05/02/nokia-6110-part-3-algorithms/`.

Appendix

Relationship with the course

In the design phase of the agent we followed the guidelines presented in Chapter 1 of the *Artificial Intelligence Fundamentals* book [3] to properly define the task environment and the agent performance measure. By defining the sub-problem of finding the shortest path from the head of the snake to the food as a search problem, we could use the search algorithms presented in Chapter 3 of the just mentioned book. In particular, we used weighted A*, defining its heuristic function to better exploit the domain knowledge.

GitHub metrics

The code has been developed on the GitHub repository available at the following link: `https://github.com/GiuliaGhisolfi/Snake`. Instructions for installation and execution are provided in the `README.md` file.

Table 1 reports the number of commits, additions and deletions made by each member of the team. Figure 1 shows the history of commits from October 23, 2022 to January 27, 2023.

Name	**# Commits**	**# Additions**	**# Deletions**
Giacomo Aru	54	42 440	3 750
Gabriele Benanti	43	5 241	2 121
Giulia Ghisolfi	125	60 579	21 808
Luca Marini	18	4 459	556
Irene Testa	62	59 224	93 458
Total	302	171 943	121 693

Table 1: Number of commits, additions and deletions made by each member of the team on the GitHub repository.

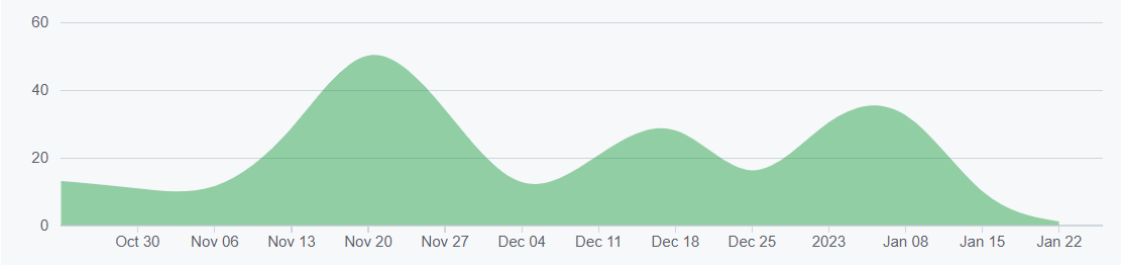

Figure 1: History of commits in the GithHub repository from October 23, 2022 to January 22, 2023.

Team contributions

All the team members equally partecipated in the development of the project. Giacomo Aru and Luca Marini mainly focused on the implementation of the Greedy Algorithm, while Giulia Ghisolfi and Irene Testa on the development of the Hamiltonian Algorithm. Gabriele Benanti worked mainly on the game interface. The code has been reviewed by all the team members.

Figures and Tables

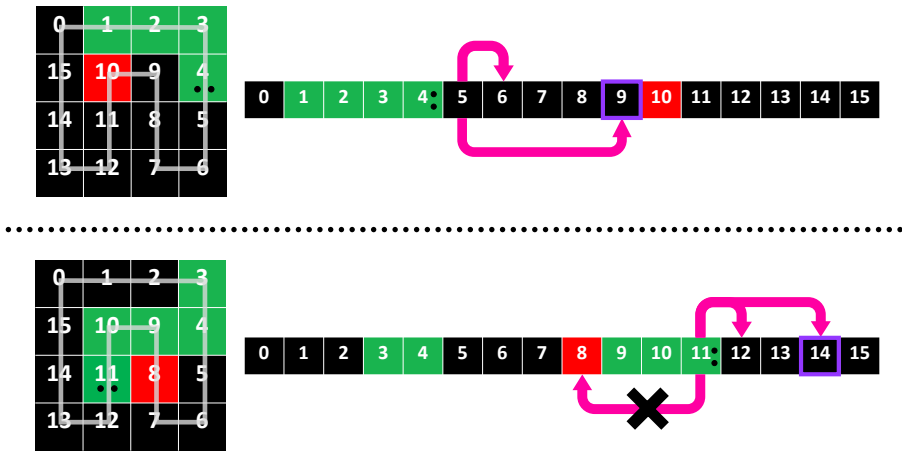

Figure 2: An illustration of the Hamiltonian Shortcuts Strategy applied to different stages of the game. On the left, the game grid in which cells are numbered according to their position in the Hamiltonian cycle; on the right the "unrolled" Hamiltonian cycle. The arrows show the possible moves the snake can make; the cell where the snake will move is marked in purple. In the example in the top panel, both the allowable moves preserve the order of the snake's body on the cycle; in the example shown in the bottom panel, instead, the move toward cell 8 does not preserve such order.

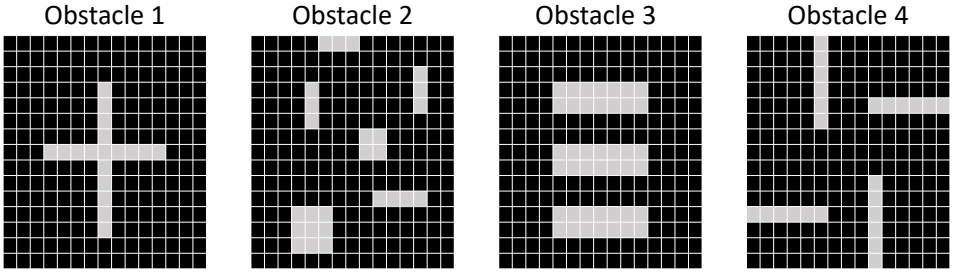

Figure 3: Obstacles configurations used to test the agents.

Greedy Algorithm Parameters					
Config. #	Heuristic	Sensibility	Safe Cycle On	Space Opt. On	A* Weights
1	2	4	0	0	[1, 1, 1, 1]
2	2	4	1	0	[1, 1, 1, 1]
3	2	4	1	1	[1, 1, 1, 1]
4	2	4	1	0	[1, 3, 3, 3]
5	2	4	1	0	[1,0.2,0.2,0.2]
6	2	2	1	1	[1,1,1,1]
7	2	3	1	1	[1,1,1,1]
8	2	1	1	1	[1,1,1,1]
9	2	5	1	1	[1,1,1,1]
10	2	2	1	1	[1,3,3,3]

Table 2: Parameters configurations tested for the Greedy Algorithm. "Heuristic" determines the heuristic to use with A, if 0 the heuristic to avoid zigzagging paths is used, if 1 the heuristic to "compress" the snake's body is used, if 2 both heuristics are used. "Sensibility" determines when the heuristics are used: if the snake length is smaller or equal than the ratio between the grid area and Sensibility, the heuristic to avoid zigzagging paths is used, otherwise the heuristic to "compress" the snake's body is used. "Safe Cycle On" determines whether to use the strategy which searches for safe cycles or to simply follow the path to the food. "Space Opt. On" determines whether to "extend" the safe path with the longest path approximation. "Weights" represents the vector of weights used with the A* algorithm. The first entry of the vector is the weight given to the path cost to the node, the second entry is the weight given to the Manhattan distance from the node to the goal, the fourth is the weight given to the additional boolean state variable which tracks if the snake turned, the fourth entry is the weight given to the state variable which stores the number of neighbors of the node. For further details, see the implementation of the algorithm and its documentation.*

Greedy Algorithm					
Config. #	Games won (%)	Time elapsed [sec]		Moves made	
		Average	Std deviation	Average	Std deviation
1	*0.00*	-	-	-	-
2	14.00	**3.02**	0.39	1639.59	152.90
3	31.90	3.06	0.49	1602.45	188.21
4	30.00	3.15	0.48	1649.72	182.76
5	30.90	3.21	0.50	1628.69	181.70
6	32.00	3.12	0.47	1618.17	175.53
7	**32.90**	3.03	0.51	**1599.31**	187.66
8	29.10	*3.22*	0.47	*1666.67*	171.45
9	31.80	3.08	0.48	1613.66	180.33
10	31.20	3.15	0.49	1639.84	182.14

Table 3: Average and standard deviation of the time elapsed and of the number of moves made to win the game on a 10×10 grid with different configurations of the Greedy Algorithm. Data obtained from 1000 executions on a Intel Core i7-10750H CPU at @ 2.60GHz.

Greedy Algorithm							
Config. #	Games lost (%)	Time elapsed [sec]		Moves made		Length achieved	
		Average	Std deviation	Average	Std deviation	Average	Std deviation
1	*100.00*	0.02	0.02	21.21	19.86	*3.78*	2.19
2	86.00	3.21	0.67	1703.14	297.56	91.09	10.13
3	68.10	3.98	0.52	1917.11	187.74	98.32	1.15
4	70.00	4.03	0.52	1948.05	192.98	98.34	1.18
5	69.10	4.10	0.49	1925.19	179.49	98.34	1.12
6	68.00	4.04	0.52	1924.59	184.99	98.33	1.14
7	**67.10**	3.98	0.52	1913.96	187.72	98.32	1.22
8	70.90	4.06	0.50	1941.32	181.77	98.28	1.15
9	68.20	3.95	0.52	1903.26	184.87	**98.36**	1.03
10	68.80	4.04	0.49	1938.90	183.12	98.32	1.15

Table 4: Average and standard deviation of the time elapsed, of the moves made and of the length achieved in lost games on a 10×10 grid with different configurations of the Greedy Algorithm. Data obtained from 1000 executions on a Intel Core i7-10750H CPU at @ 2.60GHz.

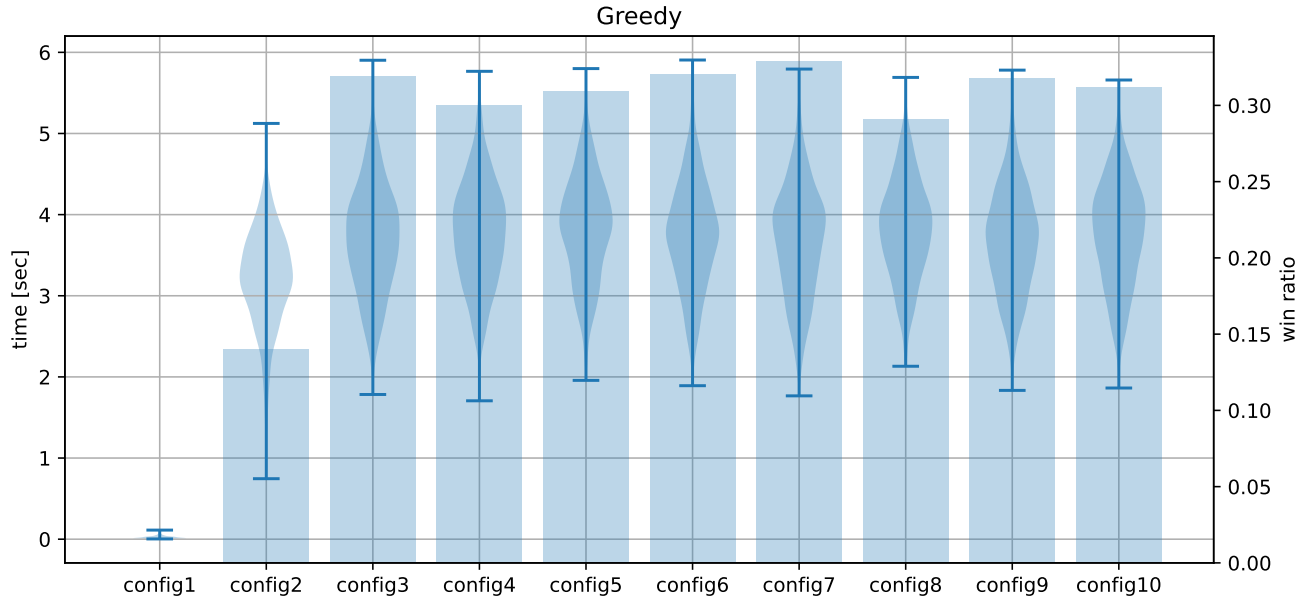

Figure 4: Total time elapsed from the begging to the end of the game and win ratio on a 10×10 grid with different configurations of the Greedy Algorithm. Data obtained from 1000 executions on a Intel Core i7-10750H @ CPU at 2.60GHz.

Hamiltonian Algorithm Parameters			
Config. #	Max Length Shortcuts	Min Length Repair	Max Length Repair
1	0.00	0.00	0.00
2	0.00	0.00	1.00
3	1.00	0.00	0.00
4	0.50	0.00	0.00
5	0.00	0.00	0.65
6	0.50	0.50	0.65
7	0.45	0.45	0.65
8	0.50	0.50	0.70
9	0.45	0.45	0.70
10	0.50	0.45	0.65
11	0.50	0.45	0.70

Table 5: Parameters configurations tested for the Hamiltonian Algorithm. "Max Length Shortcuts" is the ratio between the snake length and the grid area up to which the Hamiltonian Shortcut Strategy is used. "Min Length Repair" represents the ratio between the snake length and the grid area when the Hamiltonian Repair Strategy starts to be used. "Max Length Repair" represents the ratio between the snake length and the grid area up to which the Hamiltonian Repair Strategy is used.

Hamiltonian Algorithm				
Config. #	Time elapsed [sec]		Moves made	
	Average	Std deviation	Average	Std deviation
1	12.69	10.69	*2420.20*	142.22
2	*15.21*	11.25	2162.34	191.02
3	11.00	8.80	2120.02	240.51
4	**9.66**	8.46	1901.04	169.77
5	12.56	10.48	1921.66	160.93
6	9.97	9.30	1914.16	185.36
7	10.24	9.53	**1889.77**	197.62
8	10.44	9.55	1955.40	220.57
9	10.44	9.41	1896.60	183.08
10	10.69	9.21	1925.06	177.68
11	10.75	9.31	1961.62	168.54

Table 6: Average and standard deviation of the time elapsed and of the number of moves made to win the game on a 10×10 *grid with different configurations of the Hamiltonian Algorithm. Data obtained from 100 runs using 5 different processors. For all the configurations, in all the run, the agent won. The high standard deviations is due to the different performances of the processors used.*

Greedy Algorithm		
Game map	Game area	Games won (%)
6×6	36	**62.00**
10×10	100	30.00
16×16	256	12.00
Obstacle 1	222	26.00
Obstacle 2	214	*0.00*
Obstacle 3	198	2.00
Obstacle 4	216	5.00

Table 7: Percentage of games won with the Greedy Algorithm on different grid configurations. Data obtained from 100 executions. The percentage of games won decreases on wider grids.

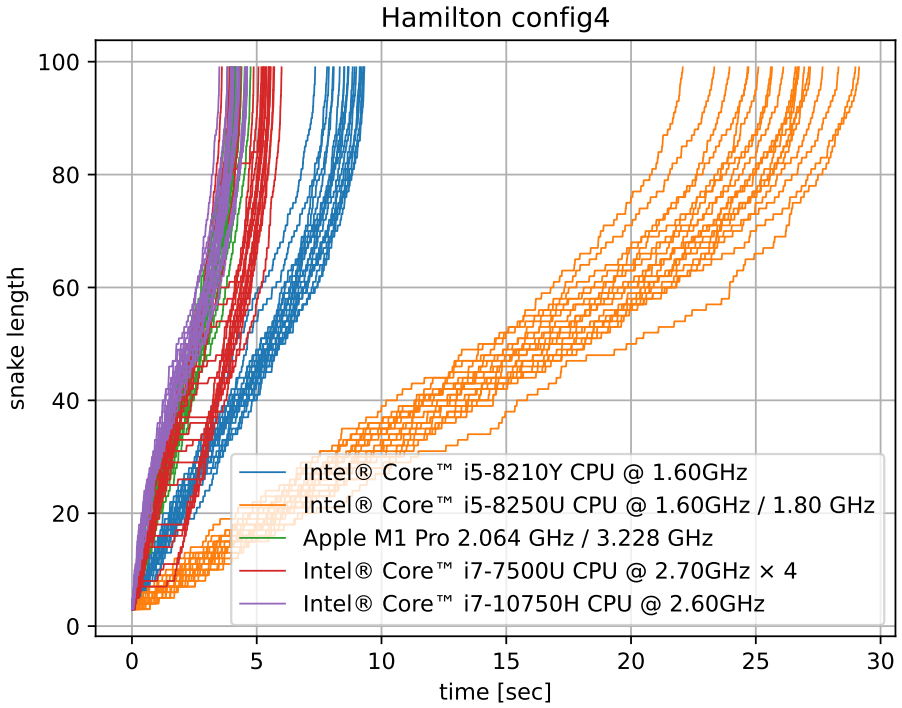

Figure 5: Snake length over time elapsed in seconds for 100 runs on 5 different processors (25 runs each) with the configuration 4 of the Hamiltonian Algorithm. Time of execution highly depends on the processor performances.

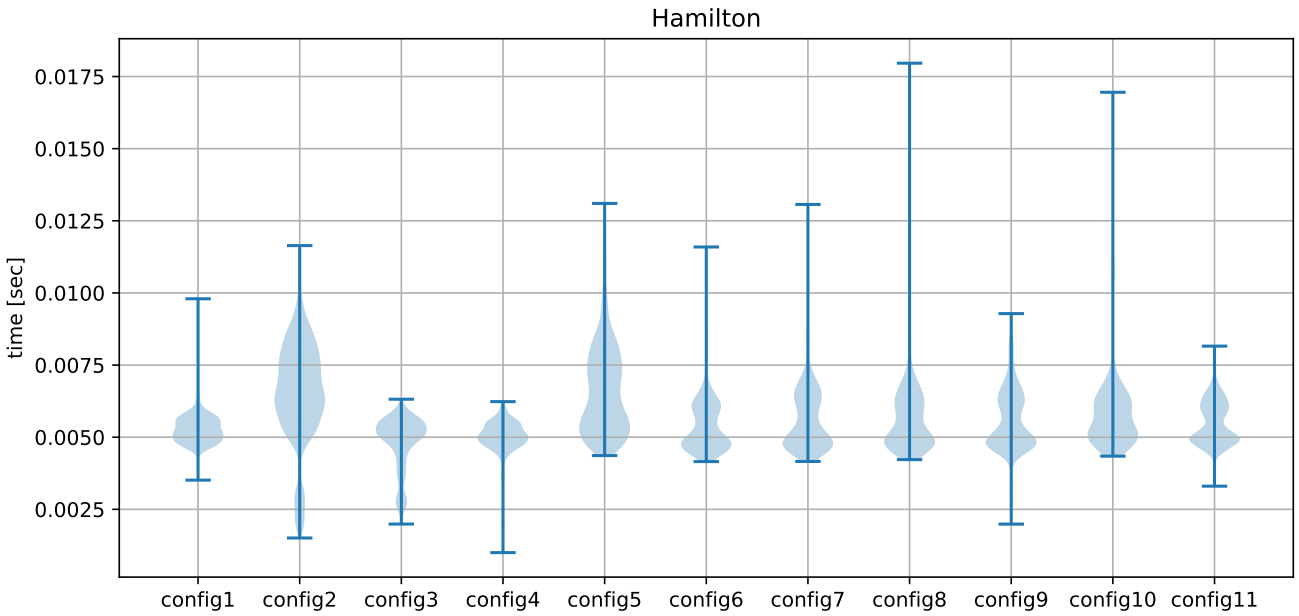

Figure 6: Time to compute each move by different configurations of the Hamiltonian Algorithm. Data obtained from 100 runs using 5 different processors.

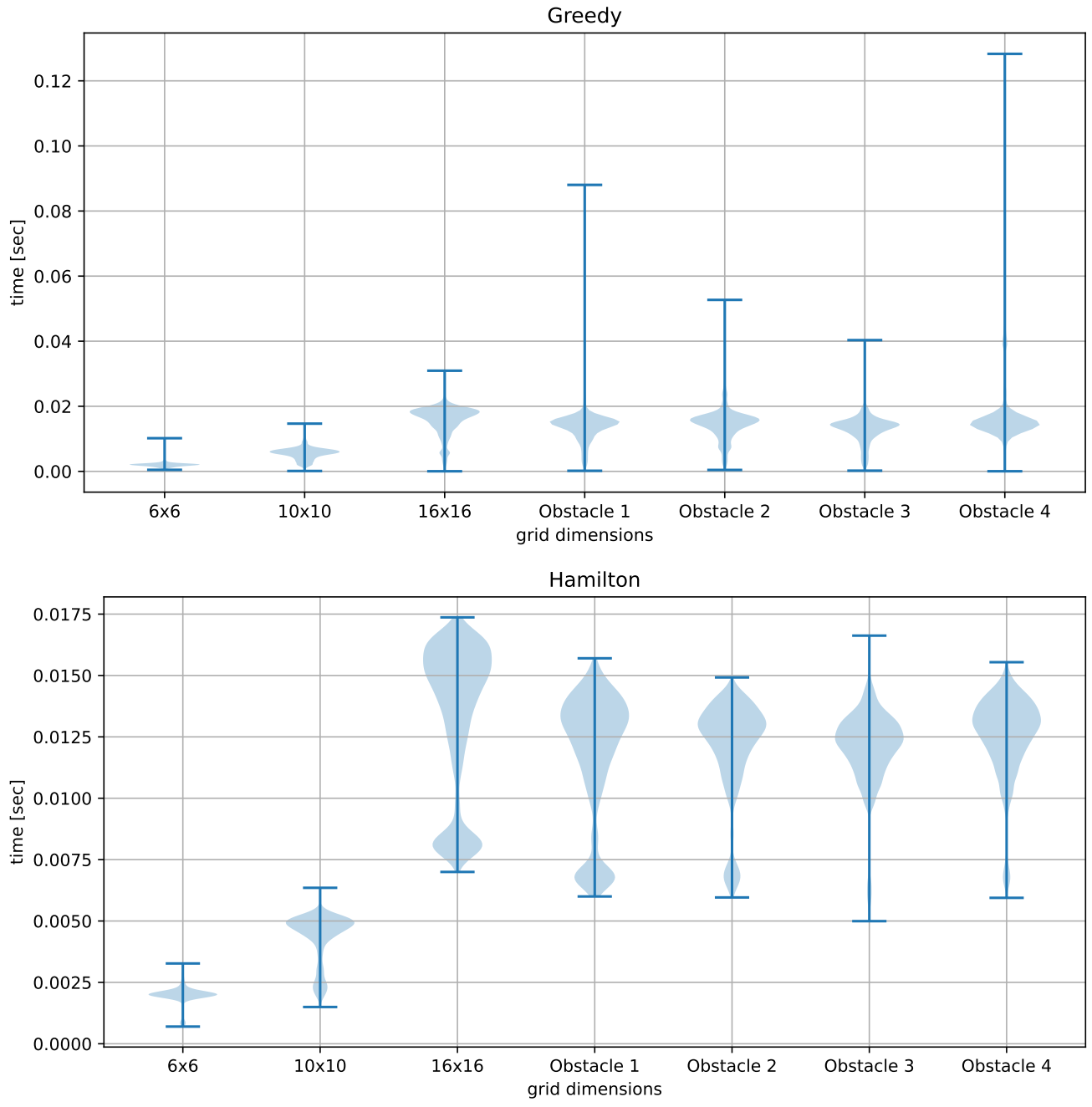

Figure 7: Time to compute each move on different game grids by the configuration 4 of the Hamiltonian Algorithm and by the configuration 7 of the Greedy Algorithm. Data obtained from 100 runs using 5 different processors.

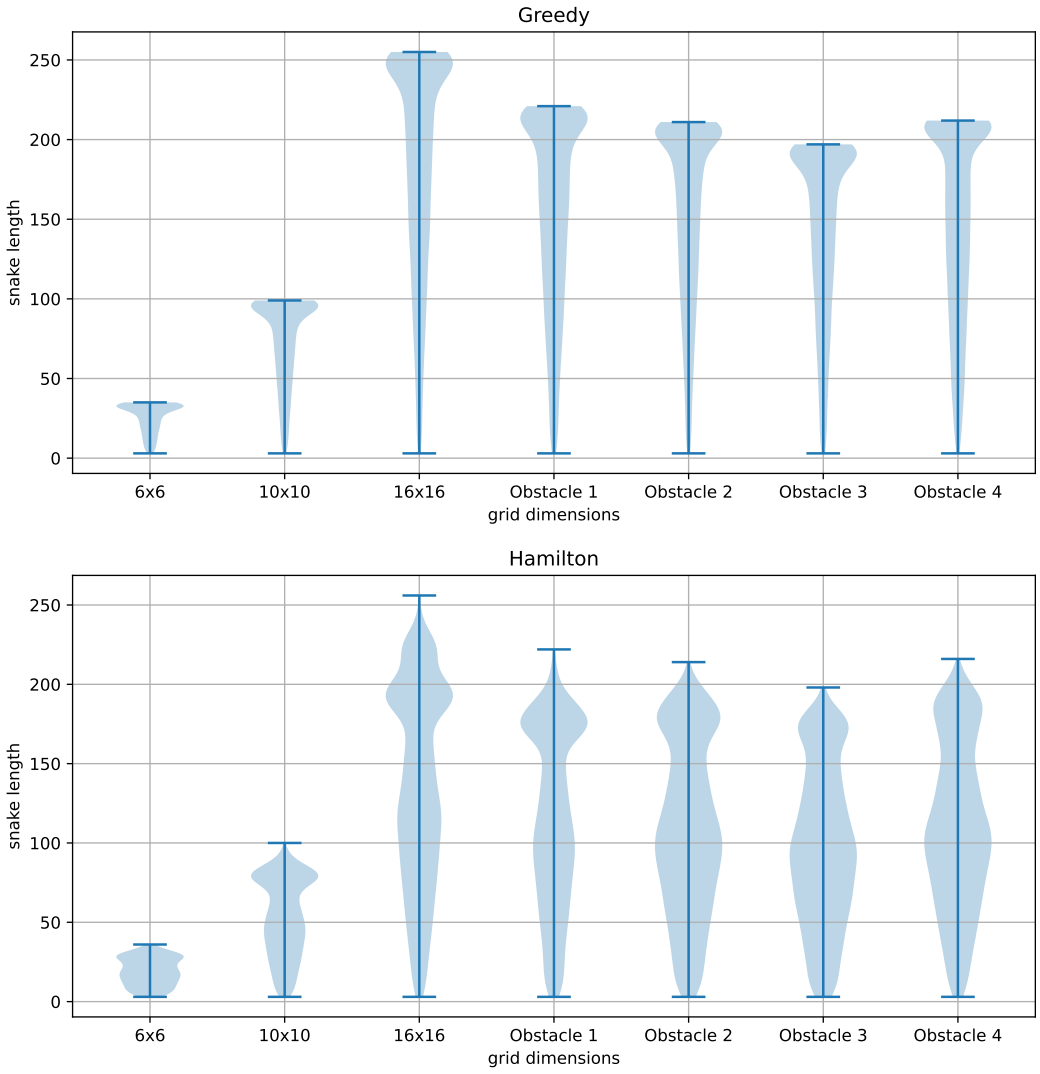

Figure 8: Snake length during the game on different grid games with the configuration 4 of the Hamiltonian Algorithm and with the configuration 7 of the Greedy Algorithm. Data obtained from 100 runs using 5 different processors.

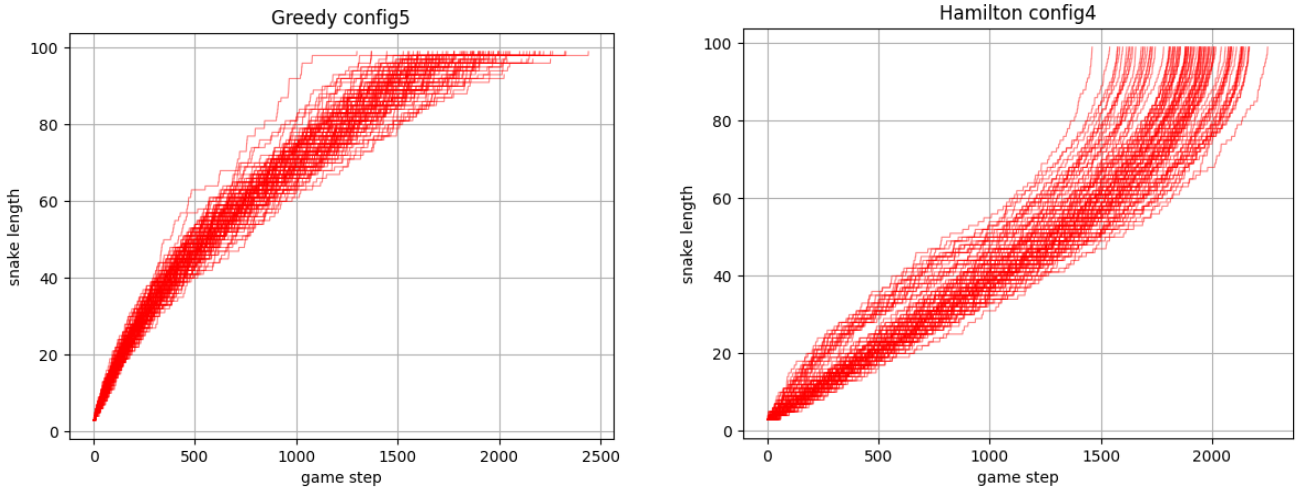

Figure 9: Snake length over moves made for 100 runs by the configuration number 5 of the Greedy Algorithm and by the configuration number 4 of the Hamiltonian Algorithm. With the Greedy Algorithm the snake grows quicker in the first stages of the game, with the Hamiltonian Algorithm, instead, it grows quicker at the end of the game.