AI homework 1

A.Y. 2023/2024

Nicola Rossi - 289927

Contents

1 Introduction

The objective of this report is to describe formal framework, analyzed problems, algorithms, heuristics and performed experiments for the first AI assignment. First, a brief description of the followed formal framework is given. Subsequent sections will be used to address:

1. the developed implementation (modules, classes, ...)
2. the problems selected for analysis
3. the implemented algorithms
4. the developed heuristics
5. the experiments conducted, to evaluate different heuristics and algorithms

1.1 Formal framework

The formal framework proposed in *Artificial Intelligence: A Modern Approach* by Russell and Norvig is used. In particular:

- for problems to be solved by (heuristic) search, is given a formal description of the problem as a 5-uple $\langle s_0,\ Actions(s),\ Result(s,a),\ Goal(s),\ g\rangle$, where:
  - s_0 is an initial state
  - $Actions(s)$ is a function mapping states to possible action from that state
  - $Result(s,a)$ is a 'successor' function, mapping a state s and an action a to another state s', which is the resulting state after the application of a to s
  - $Goal(s)$ which is the goal-test, mapping a state s to 0 (s not final) or 1 (s final)
  - g which is the path-cost function

  In this setting, since the environment is assumed to be deterministic, only actions from the agent can change the environment state and, mainly, each action has exactly one outcome. So a **solution** is a **sequence of actions**, which the agent must follow one after another in order to reach a final state from an initial state. So, in this setting, **algorithms return a sequence of actions**, not an action at a time.
- for problems to be solved by adversarial search, a formal description of the problem, slightly different from those proposed by Russell and Norving, is given. First of all, we are in a real-time setting, so search algorithms will visit the game tree only for l levels at a time. So we don't have an utility function to maximize, but an evaluation function (which, in general, does not reflect the 'real' payoff) to maximize. An adversarial search problem is described as a 6-uple $\langle s_0, Player(s), Actions(s), Result(s,a), Terminal-state(s),\ Eval(s,p)\rangle$. Most objects have a similar description to the corresponding in the single-player setting, but notice:
  - $Player(s)$ returns the player p having the move in state s
  - $Terminal-state(s)$ maps a state s to 1 if final, that is: the game has ended or 0, if the game has not yet ended
  - $Eval(s,p)$ returns a (heuristic) evaluation of the state s, under some heuristic function h, for player p

  In this setting the result of a search is not a solution (for the 'entire' problem). After the opponent's move, the agent has to re-evaluate the state and perform the action maximizing the evaluation function's value. So in this setting the **algorithms return an action at a time**, instead of a sequence of actions.

2 Implementation

The code is organized in one Jupyter Notebook and three Python modules. All the code (Notebook and modules) can be found in the attached zip. The modules are:

- **snake** (my) implementation of snake game, defining logic, moves, representation
- **chess** (pyhthon-chess) library, to manage chess games
- **stats** (my) implementation of a simple statistical utility to manage experimentation, build upon Pandas library

In the Notebook, the core classes of the implementation are defined:

- **GameInterface**, used to define the methods each Game must implement
- **ChessGame**, wrap class for chess module
- **SnakeGame**, wrap class for snake module
- **State** class, defining the basic State methods and description. For instance: representation (which depends on the Game chosen), actions() function, is_final() function, and so on...
- **Heuristics** class, used to organize and define heuristic functions used. It depends directly on the chess/snake module
- **PathCosts** class, used to organize and define path-cost functions used.
- **Algorithms** class, used to organize and implement heuristic search and adversarial search algorithms. The class uses only states (and heuristics, path costs and maximum depth, depending on the type of game), so it's completely independent from the specific game the algorithms are called to solve
- **Agent** class defining the concept of Agent, describing it using an algorithm it has to use to solve the problem/search an heuristic and other optional information such as a path cost function (for problem-solving agents) and a maximum depth (real-time adversarial search)
- **Node** class, used only by heuristic search algorithms. It's necessary to develop a pointer structure used to retrieve a solution, and to guarantee that path-cost functions' values are correctly computed.
- **Infrastructure** class, which is a convenience class whose instances are used to run experiments more nimbly. The class fields are: a list of agents (single and multi-player setting), a Game to be played by agent(s) and an instance of stats module, to collect information.

3 Problems

This section is dedicated to the formal description of the chosen problems. The main objective is to show how the chosen problem were casted to the formal framework of reference, in order to make sense of the way in which the implementation works and to show it's well defined and coherent.

3.1 Snake

In Snake game, the objective is to move the snake's head (the rest of its body will follow) in the direction of a randomly spawned apple, in order to eat it. If the snake 'eats' itself or it collides with any of the four walls, the game is over. Notice that there is no clear definition of a goal state, because the goal is basically 'survive until possible' trying to reach every time a higher score.

In a 'real-time' setting, the snake's position is updated following a certain direction, k times per second (for instance, if the game is set up to be played at 30 frame per seconds, $k = 30$). In this setting a fundamental requirement is the ability of the agent to select the next move 'very quickly'. If the agent is too 'slow' in the decision-making process, by the time the next action is selected, the snake could have collided with a wall, eaten itself or, more simply, the selected action is no longer 'useful'. Wanting to stick to the basic search algorithms studied, it's assumed the setting is not a real-time one. In other words, the snake's position changes only when a new move is chosen and applied.

Obviously, when the snake is so long that it occupies 'a lot of space', it will eventually eat itself or collide with a wall. So, can we define a state as final if the snake has eaten itself or collided? Well, actually no because in this way every time the snake eats itself or collides, a final state is reached. We are essentially admitting that losing the game is equivalent to solve the problem. So, this approach produces contradictory conclusions.

Another approach is to consider the game as made of 'tasks'. For each task a goal is to make the snake eat the apple. In the first task we have the snake (formed by only its head and tail) and an apple. If this task is *solved*, then the snake has eaten the apple and we begin with the second task. In the second task we have the snake, as was at the end of the preceding task, and a new randomly spawned apple, and so on... Eventually, we will reach a 'final' task. That is, a task where the snake eats itself or collides with a wall. In this new setting, we are able to define precisely what a final state is, that is we have a precise definition of a goal.

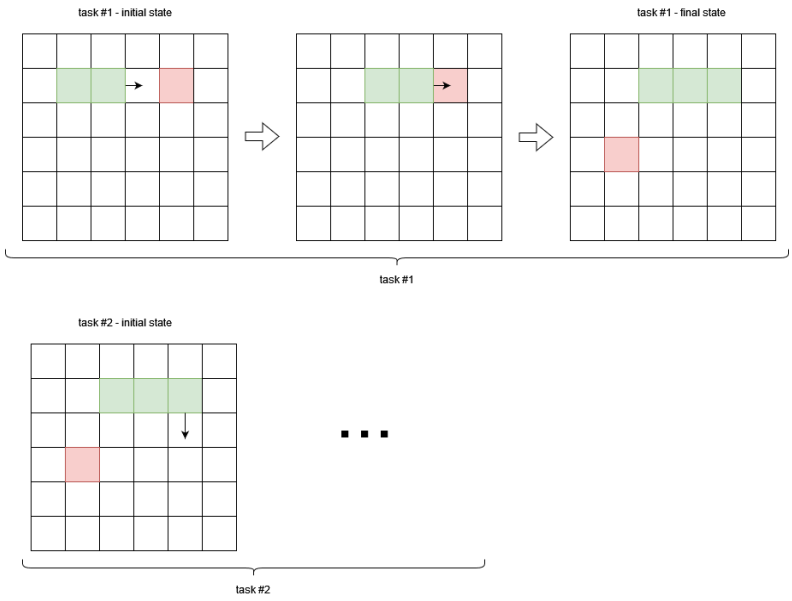


Figure 1: tasks

As a last consideration, it can happen that, following the heuristic evaluation of states, the algorithm selects as the next action to be taken, one leading to a failure (that is a game over). When such a failure

happens while searching for a solution the game simply stops and a failure is returned (more on that in algorithms section).

We are now ready to formally define Snake as a problem, or better, we are ready to define a Snake Task as a problem:

- s_0 it's every configuration in which the snake's head is not over the apple, and the snake has not eaten itself or collided with a wall (e.g. initial state of task #2, in figure). Notice that the final state of task $t-1$ is the initial state of task t, if $t > 1$.
- $Actions(s)$. Possible moves are up, down, left, right. The function returns all the legal moves from the current state. The only illegal move is the one opposite to the current direction. For instance if the current direction is up, it's not possible to move down.
- $Result(s, a)$ given a state, it computes the new snake's direction based on the specified move (e.g. from up, the direction is changed to left) and the snake's body positions are updated, shifting by one unit forward in the new direction each body piece.
- $Goal(s)$ returns 1 if the snake's has eaten the apple (that is, head position equals apple position); 0 otherwise
- g, the path-cost function in this case represents the number of moves performed to reach state a state. Therefore, g is defined to be a non-decreasing function in the depth of the game tree. In particular, if $Result(s, a) = s'$ then $g(s') = g(s) + 1$.

3.2 Chess

Chess is a two players, zero-sum, perfect information game. Since, as already noticed in the introductory section, we are in a real-time setting, the objective is to maximize the value of an evaluation function, not that of a utility function. Generally in chess, the utility values of a player are: 1 in case of victory, 0 in case of defeat and $\frac{1}{2}$ in case of a draw. The evaluation function assumes values, in general, which differs from those of utility, since the objective is to make the algorithm able to evaluate states 'quickly', evaluating the board configuration. That's why, in the formal definition of an adversarial search game problem is used $Eval(s, p)$ instead of $Utility(s, p)$. Let's now formally describe chess:

- s_0 is the initial configuration of the chess board
- $Player(s)$ returns the current player, W or B
- $Actions(s)$ returns the set of all the possible legal moves, given the configuration in state s
- $Result(s, a)$ returns a new state, in which the action a (move) is performed over (the configuration contained in) s.
- $Terminal - state(s)$ maps a state s to 1 if final, that is: the game has ended or 0, if the game has not yet ended. There are several different cases in which the game is considered ended, like:
  - checkmate
  - stalemate
  - insufficient material (either side hasn't enough material to end the game in checkmate, for instance, if W and B have both only the king)
  - ...
- $Eval(s, p)$ returns a (heuristic) evaluation of the configuration contained in state s, under some heuristic function h, for player p. Indeed, there could be configurations favouring W instead of B and vice versa, so the current player should (playing optimally) avoid actions that don't favor him, and choosing those which have the best evaluation under h.

4 Algorithms

This section is dedicated to the description of the *implemented algorithms*. The **Agent** class has two different methods, to handle two different scenarios:

1. single-player setting
2. multi-player setting

4.1 Single-player setting

Let's review some of our assumption in the single-player setting:

- the agent has no time constraint to generate a sequence of actions to solve the problem
- the environment is deterministic (from each action one and only one state is generated)
- the only entity changing the environment's state is the agent itself

These assumptions allow us to develop, in the single-player setting, a function **solve** in class Agent, which will call an heuristic search algorithm (specifying an heuristic to use) and returns a sequence of actions leading from an initial state to a final one, instead of a function returning one action at a time. The idea here is: since there are no time constraints and since the environment is modified only by the agent in a deterministic way, we can explore the game tree only one time and get a solution, instead of visit the game tree several time. Obviously, this approach on its own does not guarantee a solution is returned 'fast'. The performance will highly depend on the heuristic and the algorithm chosen.

Implemented algorithms for this setting are:

- A*
- Greedy best-first search

Both the algorithms are implemented following the pseudo-code proposed by Russell and Norving. In particular, an heuristic h must be defined as follows

$$h(s) = \begin{cases} > 0 & \text{if } s \text{ is not final} \\ = 0 & \text{if } s \text{ is final} \end{cases}$$

4.2 Multi-player setting

In this setting, instead, since the environment's state depends on the actions of more than one agent, a step-by-step approach was preferred, so in this case for each state only one action at a time is returned (also in order to better statistically evaluate the performance of algorithms and heuristics). So for multi-player settings in the Agent class a method **select_action** is defined, calling one of the implemented algorithms:

- minimax
- $\alpha - \beta$ pruning minimax

Notice that since the evaluation is based on an heuristic, the values of the states will not reflect the 'true' utility score. For instance, in chess the different outcomes have standard values: 1 for victory, 0 for defeat and 0.5 for a draw. The heuristic evaluations depends on the chess board, and will involve quantities calculated based on the pieces, position of piece, and so on, so will differ from the utility function values.
As before, the algorithms are implemented using the pseudo-code proposed by Russell and Norving. The only difference is that in my implementation randomness is added to select a move when more moves have the same evaluation.

5 Heuristics

In this section developed heuristics are presented and described. Must be emphasized that here different heuristics are described, but such functions were actually developed following the experimental results obtained by running algorithms with the first (basic) heuristics, called h_1. More info on that in the Experiments section.

5.1 Snake h_1^s

The first heuristic, takes the snake's head position at state s, let it be (x_s, y_s) and return the Manhattan distance from the head to the apple, in position (x_a, y_a), so

$$h_1^s(s) = |x_a - x_s| + |y_a - y_s|$$

Notice that the head-apple Manhattan distance is the exact distance between the head and the apple in the grid representing the snake world, so h never overestimates the cost of a solution from a state s. That is, h_1^s is **admissible** and, consequently, A* optimality is guaranteed.

5.2 Snake h_2^s

The second heuristic, is based on a little corrective applied to h_1^s. Under that heuristic, the algorithm has no 'perception' of the snake's body and the walls. This implies that the algorithm will care only about the apple's position and, consequently, it will follow the shortest path leading to the apple, even if this implies the snake has to go over itself (that is, eating itself) or has to collide with a wall. So, the new heuristic is defined as:

$$h_2^s(s) = \begin{cases} |x_a - x_s| + |y_a - y_s| & \text{if } s \text{ is not a } \texttt{failure} \text{ state} \\ +\infty & \text{otherwise} \end{cases}$$

Where a `failure` state is one in which the snake eats itself of collides with a wall. So this new heuristic highly penalizes the choice of such states.

5.3 Snake h_3^s

This heuristic was developed by a general idea and some trials-and-error. The objective was to try to reach a higher score and stabilize it, as well as have better performances. The general idea was: if the snake has space to maneuver, than it's less likely that it will kill himself in some way. Space to maneuver it's created preferring longer paths to get to the apple. Also, the space at disposal of the snake decreases as the snake's size increases.

First of all, a function f is defined, such that if d is the Manhattan distance from snake's head to the apple, then $d \leq d' \Rightarrow f(d) \geq f(d')$. This function assigns higher (worse) evaluations to states in which the apple is closer, and lower (better) evaluations to state in which the apple is 'distant'. In this way, the shortest head-apple path is never preferred and more space to maneuver will be available.

Then, has previously noted, the snake's length also plays a role, so a threshold was introduced, l representing the percentage of the area (grid) covered by snake's body (e.g. if $l = 0.1$ then the 10% of the area is covered, and if the grid is 10x10, then 10 cells are occupied by the snake).

A little malus was added to encourage the snake to stay near the walls to prevent it traps itself, cycling around with its body.

The last detail concerns the breaking of ties, necessary to guarantee good performances. It can happen that state have equal evaluations, because such evaluations are always based on the Manhattan distance and the snake has every time at most 3 possible actions, so the Manhattan distances of the resulting states differ at most by 1 distance unit. The approach used here was to introduce some randomness: randomly pick a $\epsilon > 0$ to be added to the current state's evaluation, so that each state has a slightly different evaluation, and all ties are broken.

In my setting:

- $f(d) = (1 - \frac{d}{d_{max}}) \cdot k,\ k \in [0, 1]$
- ϵ is sample from a normal distribution $N(f(d), \sigma))$
- the malus, taking value 0 if the snake is near a wall; m otherwise

Notice d_{max}. If d is measured in Manhattan distance units, then the maximum distance in a grid (board) $r \cdot c$ is $(r+c)-2$, where r represents the number of rows and c the number of columns of the grid. The parameter k in f is used to control how 'bad' lower distances have to be evaluated (or, how 'good' higher distances have to be evaluated). Finally, ϵ is picked from a Normal distribution because this ensures, with a reasonably standard deviation, that ties are broken, but states will always receive a random component in their score related to their distance, breaking ties and avoiding that states with lower distances (higher scores) would be inserted before more promising states (states with higher distances, and so, lower scores) (and vice versa).

The returned evaluation for a state is equal to $f(d) + \epsilon + malus$ if the snake's length is greater than or equal to $l \cdot (r \cdot c)$; and d otherwise (that is: if the snake is 'little').

5.4 Chess h_1^c

The very first heuristic for chess is a combination (equal unitary weights) of the material score m and king safety score k_s, defining $h_1^c = m + k_s$
Without loss of generality, the material score for a player is the sum of its pieces still on the game board, weighted by the classic weights assigned to chess pieces:

- pawn $\rightarrow$ 1
- knight, bishop $\rightarrow$ 3
- rook $\rightarrow$ 5
- queen $\rightarrow$ 9
- king $\rightarrow$ 0

If m_W and m_B are the material score, respectively, of player W and player B, then the material score $m = m_W - m_B$. m gives an idea of the advantage a player has over the other, considering the number and the significance of every piece, against those of the opponent. This implies that:

- if $m_W > m_B, m > 0$ and W is ahead of B
- if $m_W < m_B, m < 0$ and B is ahead of W
- if $m_W = m_B, m = 0$ and there is no advantage

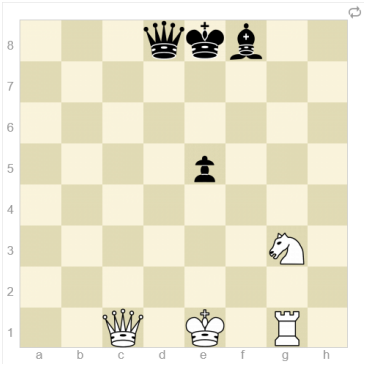
So, if $Player(s) = W$ then $h(s) = m + k_s$, while if $Player(s) = B, h(s) = -m + k_s$ because

- if $m > 0$ then W is ahead of B and B needs a 'bad' evaluation of such state
- if $m < 0$ then B is ahead of W and B needs a 'good' evaluation of such state

The king safety score k_s is the algebraic sum of:

- open files: a malus of -0.2 is given for each open file (column) adjacent to the king's one. A file is defined open is no friendly piece is in that column. In the figure, the white king has both the adjacent files opened, the black one both 'closed'.
- attacking pieces: a malus of -0.3 is added for each opponent's piece attacking the king
- castling: a bonus of 0.2 is given if the player's king can castle

Summing up all these components, given an estimate of king's safety. The general idea is: try to make the players more aware of the king and possibly avoid configuration in which the kings are very unsafe, which will lead to low-quality games.

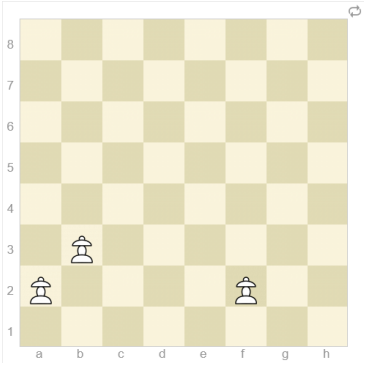


5.5 Chess h_2^c

The second heuristic was en extension of the previous one, thought to develop the game in a more tactic way: not only the material score and the king's safety are considered, but also some position scores, which potential have implications even in the endgame. For instance, isolated pawns become weaker and weaker as the game approaches the end.
To $h_1^c(s)$ is added (algebraically):

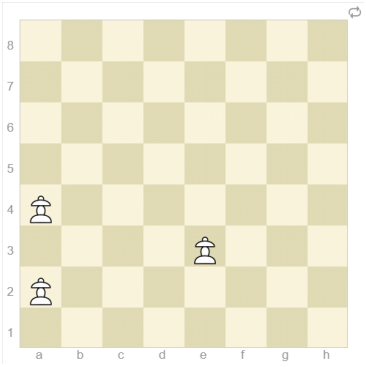
- a malus of -0.5 for each isolated pawn
- a malus of -0.5 for each doubled pawn
- a bonus of 0.2 for each current player's piece in the center of the board (D4, E4, D5, E5)

In figure, pawn in f2 is isolated, while pawns in a2 and b3 not.



Isolated pawn are usually considered a weakness, since they can't be protected by other pawns: this mean that in order to protect the pawn, a piece must be used and it can't at the same time be used to make threats to the opponent's pieces. Also, the square in front of an isolated pawn can be used by the opponent as an outpost, since every piece placed that square will be immune to pawn attack.

In figure, pawns in a2 and a4 are doubled, while pawn in e2 not (but it's isolated).



Doubled pawns are usually considered a weakness, since standing one ahead/back to the other, they can't protect each other.

Finally, the center control. There are several reasons why controlling the center is an advantage: pieces at the center of the chess board have 'higher mobility', thus can make more threats to the opponent, participate in actions in both sides of the board, and so on... .

6 Experiments and Results

6.1 Snake experiments

The main objective of these experiments was to try different heuristics with different algorithms, in order to

- evaluate heuristics quality, in terms of the effect it has on performance of the chosen algorithm
- develop new heuristics, based on the experimental results of other tried heuristics and chosen algorithm
- conduct experiments to have a better understanding of the theoretical results

6.1.1 First trial

The very first trial was: A* with heuristic h_1^s. The idea was: since, as previously stressed, h_1^s never overestimates the head-apple distance, the A* returns optimal solutions. In this case, optimality refers to the number of moves the snake does to reach the apple.

	TIME	# EXPLORED	# MOVES	SCORE
mean	103.125145	6839.900000	79.400000	10.500000
std	211.648598	8936.404254	43.133385	4.949747
min	0.015622	152.000000	12.000000	1.000000
25%	0.776459	803.750000	66.750000	9.000000
50%	4.562526	2387.500000	71.000000	9.500000
75%	79.972368	10718.750000	97.000000	13.250000
max	680.492659	28702.000000	146.000000	18.000000

From the experiment (sample size $n = 10$) we notice that the average score is very low, and its small standard deviation suggests that it's relatively centered around its mean. So the first trial was not very encouraging. On the other hand, it's also important to notice that with h_1^s the agent is not aware of walls and other body pieces. In fact, 8/10 times the game over cause was snake eaten itself, and the remaining was due to a wall collision.
Another interesting number is the mean number of explored node and its standard deviation: it can be seen that the second is pretty high, suggesting the number of explored nodes is highly variable. Finally, we can see that the ratio

$$\frac{\bar{\mu}_{moves}}{\bar{\mu}_{score}} \approx 7.6$$

6.1.2 Second trial

In order to add some awareness of walls and body pieces (in general, obstacles), h_2^s was introduced. The idea was very simple: if the snake encounter an obstacle, poorly evaluate that state. It seemed a promising approach, at least to guarantee higher scores, while maintaining 'low' the number of moves.

	TIME	# EXPLORED	# MOVES	SCORE
mean	2420.980072	38211.0	317.0	41.0
std	NaN	NaN	NaN	NaN
min	2420.980072	38211.0	317.0	41.0
25%	2420.980072	38211.0	317.0	41.0
50%	2420.980072	38211.0	317.0	41.0
75%	2420.980072	38211.0	317.0	41.0
max	2420.980072	38211.0	317.0	41.0

A* 'equipped' with h_2^s was very slow, it explored a huge number of nodes. The initial idea was to collect a sample of 10 runs. The simulation started taking too much time, so it the results obtained until that time

point were collected. Just one run took about 40 minutes! On the other hand, a score of 41 was reached. Notice also that the ratio

$$\frac{\bar{\mu}_{moves}}{\bar{\mu}_{score}} \approx 7.7$$

So comparable to the previous one.

6.1.3 Third trial

Then a paradigm shift was attempted. The question was: is it really necessary to have an optimal solution in terms of number of moves? Do we really need to care this much about the number of moves? Maybe relaxing optimality, we can achieve a better score, at the cost of making more moves. It does indeed make sense: ideally, if the snake uses some moves to create a 'safe' configuration of its body, it can potentially stay alive more time and eat more apples. Also, the objective is to have better performances, i.e.: exploring less states.
With these ideas in mind, was first tried a Greedy best-first search approach, 'equipped' with h_2^s. The results were very encouraging:

	TIME	# EXPLORED	# MOVES	SCORE
mean	67.825056	5612.00000	367.300000	40.300000
std	135.802270	6906.46919	160.362125	13.224976
min	0.012261	175.00000	152.000000	20.000000
25%	0.104052	434.75000	236.250000	32.250000
50%	3.152152	1986.00000	364.500000	40.000000
75%	48.173708	9673.25000	501.250000	49.250000
max	422.046822	19901.00000	591.000000	62.000000

We start noticing that the average time to solve a task (reach an apple) is about 1.5 time lower than the average time for A* with h_1^s and the standard deviation suggests that the time needed in this setting is more centered around the mean.

Also, since time needed to solve a task highly depends on the number of explored states, we can observe as well that the average number of moves (as well as its standard deviation) is lower than h_1^s, A* setting. Now: consider the ratio

$$\frac{\bar{\mu}_{explored}}{\bar{\mu}_{moves}}$$

That is a (very) rough estimate of the number of state to be explored to select a move. In this case, the ratio is ≈ 15.3, while in the other setting is ≈ 86.2. So for each move, with this approach, we need to explore far fewer states. On the other hand, in this case

$$\frac{\bar{\mu}_{moves}}{\bar{\mu}_{score}} \approx 9.2$$

But, considering also that the average score is now ≈ 40, that is about 4 times higher than h_1^s, A* setting, we can conclude that Greedy,h_2^s is a promising approach, both for performance and score. Concluding, applying the Greedy allows the agent to solve the problem faster and with higher scores, compared to A*, at a cost of a non-optimal solution (in terms of number of moves), and these experimental results match the theoretic ones.

6.1.4 Refining and tuning

The heuristic h_3^s was thought and introduced to have better results, in terms of score and time. The heuristic, as the previous case is used with the Greedy algorithm. Follows the experimental results for the setting:

- $f(d) = (1 - \frac{d}{d_{max}}) \cdot 0.2$
- ϵ is sample from a normal distribution $N(f(d), \sigma))$

- the malus, taking value 0 if the snake is near a wall; 5 otherwise
- $l = 0.10, r = 10, c = 10$

	TIME	# EXPLORED	# MOVES	SCORE
count	10.000000	10.000000	10.000000	10.000000
mean	1.809116	2794.300000	725.900000	41.800000
std	1.527608	1193.865715	231.410386	11.448435
min	0.082422	405.000000	205.000000	17.000000
25%	1.073290	2338.500000	658.750000	39.250000
50%	1.430845	2822.500000	771.500000	46.000000
75%	1.795572	3504.000000	867.250000	48.750000
max	5.446537	4292.000000	1000.000000	53.000000

We can observe that $\bar{\mu}_{time} \approx 1.8$, which is ≈ 37.6 time faster than Greedy equipped with h_2^s! Also, the st.dev. suggests time needed is very close to the mean, so it's more stable than h_2^s. Obviously, with such times, we expect also that the number of explored states is very contained and it's indeed the case.
Finally, let's observe the score: this heuristic allowed to achieve a slightly better average score and make the score more stable (look at the st.dev.), with respect to h_2^s. Also, comparing the median score between h_3^s and h_2^s it can be appreciated the improvement in the score.

6.2 Chess experiments

Also in this case, the main objective of the experiments was to try different heuristics with different algorithms, in order to

- compare algorithms performance under the same and under different heuristics
- check the validity of theoretical results

It would also be interesting develop, as for the Snake game, new heuristics based on experimental results, but chess is a game much more complex than snake and building new heuristics also requires experience as chess player.

Another important point is that the computational resources required to run the simulations in this case are substantial, so very limited depth are chosen (maximum 3), and samples sizes are very small.

Just as a matter of notation convenience, the trials are represented as tuples $\langle h_W, alg_W, d_W, h_B, alg_B, d_B \rangle$, representing the heuristic, algorithm and depth used by W and B player.
The first part of this section is used to check the experimental results against the theoretical ones. The second part is used to compare the heuristics.
The first experiments tested, using the same heuristic (h_1^c) as benchmark, the different performance of the algorithms.

6.2.1 $\langle h_1^c,\ \alpha\beta,\ 1, h_1^c,\ minimax,\ 1 \rangle$

On a sample of size $n = 10$ games, both W and B won 5 of them. Here it's interesting to notice that the the average number of explored state was $\bar{\mu}_{explored} \approx 26831$, st. dev. ≈ 21630.4 for W, while for B $\bar{\mu}_{explored} \approx 25961$, st. dev. ≈ 22458.5, so in this case we have comparable numbers in average number of explored nodes, and standard deviations are comparable as well. Considering also other statistics (below), we can conclude that at depth 1, $\alpha\beta$ has similar performance to $minimax$ which make absolutely sense, since the number of nodes grows exponentially in the depth. That is, we expect $\alpha\beta$ to out perform $minimax$ at higher depths.

	# EXPLORED
mean	26831.000000
std	21630.416568
min	5979.000000
25%	9008.250000
50%	19685.500000
75%	40408.750000
max	64977.000000

statistics $\alpha\beta$

	# EXPLORED
mean	25961.300000
std	22458.514149
min	5951.000000
25%	9024.250000
50%	19598.500000
75%	37611.250000
max	71765.000000

statistics $minimax$

6.2.2 $\langle h_1^c,\ \alpha\beta,\ 2, h_1^c,\ minimax,\ 2\rangle$

Then, the depth was increased by 1, and still with one sample of size $n = 10$. Here we notice the different order of magnitude in the average number of explored nodes between $\alpha\beta$ and $minimax$. As expected from the theoretical results, $\alpha\beta$ completely outperforms $minimax$. Consider the ratio

$$\frac{\bar{\mu}^{minimax}_{explored}}{\bar{\mu}^{\alpha\beta}_{explored}} \approx 9.4$$

So for each state explored under $\alpha\beta$, $minimax$ explores (roughly) 9.4 states.

	# EXPLORED
mean	304442.300000
std	241215.212925
min	17837.000000
25%	84289.250000
50%	317017.000000
75%	453770.250000
max	648508.000000

statistics $\alpha\beta$

	# EXPLORED
mean	2.862208e+06
std	2.528914e+06
min	7.674500e+04
25%	3.952745e+05
50%	3.109740e+06
75%	4.308936e+06
max	7.187422e+06

statistics $minimax$

Having a look also at the other statistics, we can definitively conclude that the experimental results match the theoretical ones.

Then, experiment to evaluate the heuristics were performed:

6.2.3 $\langle h_1^c,\ \alpha\beta,\ 3, h_1^c,\ \alpha\beta,\ 3\rangle$

This trail was conducted to test how the agent performed, playing under the same starting conditions, with h_2^c heuristic. On a sample of size $n = 10$ games, it was observed that the average game time was $\bar{\mu}_{time} \approx 5026.7$ s, with standard deviation ≈ 3766.5 s, while the average number of moves for a game $\bar{\mu}_{moves} \approx 199$, with standard deviation ≈ 160.
In particular, this huge number of moves can suggest that h_1^c makes the players more focused on strategy: prioritize material advantage, focus less on decisive outcomes.
W won 5 games, B won 4 games and one game resulted in a draw, suggesting that agents are on the same level, as one would expect.

W had $\bar{\mu}_{explored} \approx 5704807$, st.dev. ≈ 3505694. B has $\bar{\mu}_{explored} \approx 4530755$, st.dev. ≈ 2671943.

6.2.4 $\langle h_2^c,\ \alpha\beta,\ 3, h_2^c,\ \alpha\beta,\ 3\rangle$

This trail was conducted to test how the agent performed, playing under the same starting conditions, with h_2^c heuristic. On a sample of size $n = 10$ games, it was observed that the average game time was $\bar{\mu}_{time} \approx 5668.7$ s, with standard deviation ≈ 3303.8 s, while the average number of moves for a game $\bar{\mu}_{moves} \approx 112$, with standard deviation ≈ 73. Here we immediately notice that the average number of moves is way lower than the previous setting. Also, W and B both won 5 games, suggesting that the no agent has an advantage over the other (as expected). At the same time, it's important to notice that no game concludes in a draw. This can suggest, along with the lower average number of moves with respect to h_1^c, that h_2^c makes the agents play more tactically, instead of strategically, and so is capable of creating positions with **decisive outcomes**.

W had $\bar{\mu}_{explored} \approx 6328870$, st.dev. ≈ 3021287 and median ≈ 5490012. In particular, since the st.dev. is very high, as before, the median was considered. B has $\bar{\mu}_{explored} \approx 5518376$, st.dev. ≈ 3031831 and median ≈ 4758348. So here it can be noticed that the number of explored states is highly variable, but the medians (robust to outliers) are comparable, suggesting that in the 'majority' of the scenarios, both players explore a comparable number of states. Comparing this numbers with the relative one in the previous setting, it's also possible to notice that h_2^c makes the algorithm explore more states than h_1^c does. The relationship between explored states under h_1^c and h_2^c will be analyzed in more detail in the last setting's analysis.

6.2.5 $\langle h_2^c,\ \alpha\beta,\ 3, h_1^c,\ \alpha\beta,\ 3\rangle$

This trial was critical to evaluate the differences of h_1^c and h_2^c and their effect on the games. On a sample of size $n = 11$ games, the average time for a game $\bar{\mu}_{time} \approx 4907.7$ s, with standard deviation ≈ 3498.4 s, while the average number of moves for a game $\bar{\mu}_{moves} \approx 114$, with standard deviation ≈ 94. We can notice, with respect to the previous cases, that the average number of moves is close to the second setting, but with higher variance.

W won 7 games, B 3 and one resulted in a draw, suggesting the heuristic h_2^c can give a slightly better way to evaluate the board and make better decisions. Additionally, this interpretation is also supported by the previous analysis, since h_2^c is probably capable of make the agent more prone to make decisive moves. Also recall that some choices are randomized so it's difficult, with this sample size, have a good understanding on what's going on: maybe some B's victory is just a matter of case or the randomness in the algorithm could have lead B to make a choice resulted in an advantage in the endgame.

Then the number of explored nodes was investigated: W had $\bar{\mu}_{explored} \approx 5649410$, st.dev. ≈ 3721135 and median ≈ 7104019. In particular, since the st.dev. is very high, as before, the median was considered. B has $\bar{\mu}_{explored} \approx 4283651$, st.dev. ≈ 2721345 and median ≈ 5220322. So, looking at the medians, we can actually say the W explores more states than B. So further investigation was conducted:

- W explored, on average, ≈ 105258 states to select a move (st. dev. ≈ 51308.8) and needed (roughly) ≈ 49.5s per move (st. dev. ≈ 26.6s).
- B on the other side,B explored on average ≈ 81940 states to select a move (st. dev. ≈ 38305.6) and needed ≈ 35.1s per move (st. dev. ≈ 18.3s)

At this point the question was: both W and B are playing using the most efficient implemented algorithm and are exploring the game tree at the same depth, so why W actually explores more states? The answer lies in the **branching factor**.
Here it is calculated a very (very) approximate estimate of the **effective branching factor** b^*, defined as the number b^* such that

$$N + 1 = 1 + b^* + \ ... \ + (b^*)^{d-1} + (b^*)^d$$

where N is the average number of explored states per move and d is the explored depth, assuming to be exploring a full, finite tree of depth d and number of child per node constant and equal to b^*.

For h_2^c player, W, the estimate of $b^* \approx 47$, while for h_1^c player, B, is ≈ 43. These values are not very different, but it must be taken into account that the st.dev. of the average number of moves is very high, which makes branching factor very unstable. Trivially, the randomness and the uncertainty of chess can easily lead to very different number of moves/state explored and, so, to very different branching factors from game to game. But, anyway, since the estimate of the W's branching factor is higher than B's one, we get an insight that h_2^c, considering more details to evaluate a state, makes the player explore more nodes due the the higher number of child nodes at each level. On the other hand, h_1^c makes the player more 'unconscious': its evaluation does not consider all the details considered by h_2^c, leading to a lower number of child nodes in each level. Even if there is no strong evidence of these assertion, from these experimental results, we can conclude they are plausible and consistent with the different logics of the two heuristics and with the previous experimental results on each heuristic.