

Probability in Game Design

Abstract

This tutorial introduces the first notions of probability, graph theory, and AI which are relevant for modelling simple games. For graphs, we will see the formal definition of graphs, multi-graphs, and finite automata, while for AI we will list some of the key set of features providing a broad classification of games, as well as describing their problem. Last, we are going to see how these abstract notions can be put into practice in C#, thus providing a practical explanation of what was theoretically illustrated during the lecture.

1 Introduction

In this first tutorial, we want to provide the minimal knowledge required to model a simple game. The game we are considering is defined as follows:

The game is composed of 4 rooms: s , s_1 , s_2 , and s_3 . The game starts from the first room, s ; the goal of the game is to reach the last room, s_3 , after picking the correct doors. The player could move from one room to the other via some doors: each room (except from the final one, s_3) has six closed doors, all in the same wall, and they all lead to the next door (Figure 1): i.e., all the doors in s lead to s_1 , all the doors in s_1 lead to s_2 , and all the doors in s_2 lead to s_3 . A player could never turn back and change his previous decision. A door might be also locked. Last, only one of the six doors within a room is the correct one.

By comparing this simple game with the other ones that we know, we identify a small number of properties along which (video)games can be characterized. These properties will also help to better design the game as the main features to be implemented will be circumscribed.

- **Discrete** vs. **Continuous**: this distinction applies to the way the time is handled within the game, as well as when the actions of the player are performed. In-person shooters or a car races are usually described as a continuous-state and continuous-time problem: both the speed and the location of the players sweep through a range of continuous values that might change through time. On the other hand, our game fits a discrete description, as it might be organized in turns (e.g., pick a door, go to the next door, and so on).



Figure 1: Graphical representation of a non-final room from the game, containing six doors, which all lead to the same next room.

- **Single Agent vs. Multi-Agent:** while an agent playing PONG is in a two-agent game, and therefore in a multi-agent scenario, in our simple game we have a single agent. This terminology is broad enough to also include *cooperative games*. If the game is non cooperative, Single Agent is often referred as PvE, and multi-agent as PvP.
- **Known vs. Unknown:** this distinction refers to the agent's state of knowledge about the rules of the game. In a known environment, the outcomes (or outcome probabilities) for all actions are given. If the environment is unknown, the agent must learn "how it works" in order to make good decisions. For the moment, we will consider only *known games*, and we will leave the unknown ones while dealing with Reinforcement Learning topics.
- **Episodic vs. Sequential:** in an episodic environment, any current decision of a player does not affect the right set of actions to be performed in the subsequent state of the game while, in sequential environments, the current decision could affect all future decisions. Therefore, we could differentiate between "trivial gambling" (where the win or loss is completely or nearly random) and "strategic" games. This tutorial will show that slight changes in the game's assumptions will make an initially episodic game into a sequential, thus testing the player's cunning.
- **Fully observable vs. Partially observable:** the game is fully observable if an agent's sensors give it access to the complete state of the game at each point in time. While the game of chess is a fully observable game, as the board is completely visible all the time, our game is partially observable, as the agent will have no previous knowledge concerning the correctness of a given door.

Under these assumptions, we can implement a game, where we have a single agent performing *some actions* over the observed world. The program will contain the complete information of the world, and the agent *will act and receive stimuli as a response*. At this stage, we identified the main properties of our game, but we have not modelled how the single agent should interact with the world. Any game, as any AI problem, can be reduced to five components:

- The **state space** V describing the game. In our game, we have the four rooms.
- The **initial state** that the player starts in. In our game, the player always starts from s with all the doors being un-selected and closed.
- The players can perform **actions** to possibly move from one state to the other. In our game, the agent could only choose a door to open or, if he reaches a wrong game configuration, must only backtrack to a previous state of the game.
- A **transition model** describes whether an action has an effect over the change of status of the game. In our game, each chosen unlocked door always leads to a room: e.g., $s \xrightarrow{5} s_1$.
- The **goal tests** determines whether the player reached a winning configuration of the game. If the player reaches a state that both doesn't allow it to move and doesn't satisfy the goal test, then he reached a losing state. In our game, the goal test is met if s_3 is reached after selecting all the correct doors since the last time the player visited node s .

By changing the conditions on the door, we will see how even a very simple game could end up with very different possible running times, as well as it might come with different chances of being solved. In addition to that, we want to model the possible doors and rooms within the game, thus modelling the set of possible configurations of the game. For assessing the first part, we need some preliminary notions of probability theory while, for modelling the second, we need to define graphs.

2 Background

2.1 Introduction to Probability Theory and Pseudorandom Number Generators in C#

Probability theory is a way to model mathematically the chance: we refer to the phenomenon that we are observing as **experiment**, and we assume that a **random variable** X associated to this

experiment could provide multiple possible outcomes; in particular, $I(X)$ denotes the set of all the possible values that could be returned by a variable X .

Since most applications prefer not to use known and sampled random number sequences from natural events for both predictability and safety reasons, most applications use pseudo-random number generators, where random numbers returned by X algorithmically generated by a decidable program¹. The default PRNG for C# is given in the `Random` class declared in the `System` namespace; the constructor of such a class accepts a number, called **seed**: the seed guarantees a generation of the same sequence of number in each possible run of the program. This feature guarantees that all the experiments are repeatable, thus easing the debugging process of programs that are highly dependent on a sequence of unforeseeable values.

As the `NextDouble()` method of such a class always return a floating point number with double precision between 0 and 1, 1 excluded, we could always transform the output of such a function so to return values within an arbitrary interval $[a, b)$ by applying the following transformation:

$$a + r.NextDouble() \cdot (b - a)$$

On the other hand, if we want to return integers within the same interval, we can invoke the `Next` method with the following arguments: `Next(a, b + 1)`. The following class, provided in our source code, implements the aforementioned utilities, where the range boundaries $[a, b]$ are specified:

Listing 1: `utils/UniformRandom.cs`

```
1 using System;
2 namespace ConsoleApp2.utils {
3     class UniformRandom {
4         double a, b;
5         public UniformRandom(double a = 0.0, double b = 1.0) {
6             this.a = a; this.b = b;
7         }
8
9         public double nextReal(ref Random r) { // I(r) = [a, b)    I(r) ⊆ ℝ
10             return a + r.NextDouble() * (b - a);
11         }
12         public int nextInt(ref Random r) { // I(r) = [a, b]    I(r) ⊆ ℤ
13             return r.Next(((int)Math.Round(a)), ((int)Math.Round(b))+1);
14         }
15     }
16 }
```

If you want to return doubles where both the range boundaries are included, we could replace the `NextDouble` method with `Sample`.

We denote a random variable E returning either 1 or 0 as **event**; such random variable returns 1 when the event associated to E happens, and 0 otherwise. E.g., given each possible value i in $I(X)$ ($i \in I(X)$), we can define $|I(X)|$ possible events $X = i$, where each event $X = i$ returns 1 when X returns i and 0 otherwise. For these events E , we are interested in assessing the **probability** $\mathbb{P}(E)$ which, from the frequentist point of view, can be interpreted as the ratio between the number of the possible cases when E happens over the number of the total possible cases. If the experiment X is not rigged and/or each associated event $X = i$ is equiprobable, then we can provide a simplified definition of $\mathbb{P}(X = i)$ as follows:

$$\mathbb{P}(X = i) = \frac{1}{|I(X)|}$$

In particular, given that the values generated by `Random` are guaranteed to be equiprobable, we might also prove that the transformations in `UniformRandom` does not alter this condition².

Given the set of all the possible values $I(X)$ and the probabilities $\mathbb{P}(X = i)$ for each expected value i in $I(X)$, we can also obtain the **expectation** (i.e., a probabilistically weighted-mean expected value) $\mathbb{E}(X)$ as follows:

$$\mathbb{E}(X) = \sum_{i \in I(X)} i \cdot \mathbb{P}(X = i)$$

¹See https://en.wikipedia.org/wiki/Pseudorandom_number_generator for more details on PRNGs.

²See also https://en.wikipedia.org/wiki/Continuous_uniform_distribution.

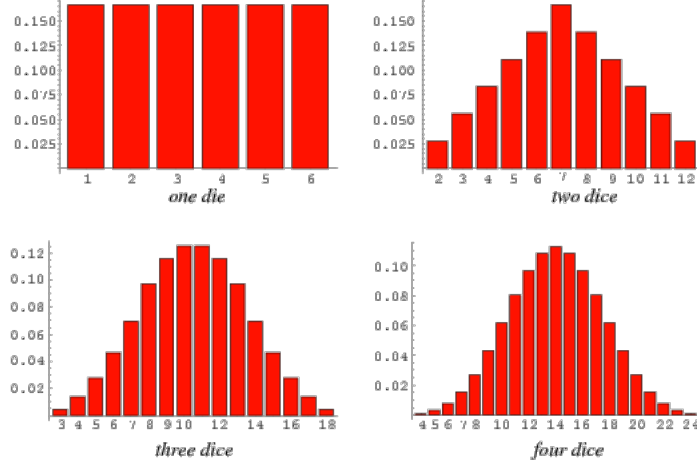


Figure 2: Probability distributions associated to the values generated by summing up the outcomes of either one or multiple dices. The plots in the first row refer to Examples 1 and 2. Source: *MathWorld*

Example 1. Let us consider “the outcome of rolling a single die” as our experiment of interest, and let us D denote a possible outcome of such experiment: we would have $I(D) = \{1, 2, 3, 4, 5, 6\}$. If we assume that the dice is not rigged, then the outcome of each experiment is equiprobable, and we obtain $\mathbb{P}(D = 1) = \mathbb{P}(D = 2) = \dots = \mathbb{P}(D = 6) = 1/6$. Then, we can compute the mean expected value as follows:

$$\mathbb{E}(D) = \sum_{i \in I(D)} i \cdot \frac{1}{6} = \frac{1}{6} \sum_{i=1}^6 i = \frac{1}{6} \cdot \left[\frac{i(i+1)}{2} \right]_{i=6} = 3.5$$

□

Returning now to our use case, please observe that the probability of choosing the i -th door out of 6 corresponds to the probability of a dice returning i after being rolled.

Example 2. Let us now define the random variable $Y := X + X$ denoting “the sum of the outcome of rolling two distinct dices”; now, we have $I(Y) = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$, but each event $Y = i$ is not equiprobable, as $Y = 2$ could only happen when both dices have $X = 1$, while $Y = 7$ happens for all of these cases that are a subset of $I(X) \times I(X) = I^2(X)$: $\{(3, 4), (4, 3), (6, 1), (1, 6), (2, 5), (5, 2)\}$. Therefore, we would have:

$$\begin{aligned} \mathbb{P}(Y = 2) &= \mathbb{P}(Y = 12) = 1/36 & \mathbb{P}(Y = 5) &= \mathbb{P}(Y = 9) = 4/36 \\ \mathbb{P}(Y = 3) &= \mathbb{P}(Y = 11) = 2/36 & \mathbb{P}(Y = 6) &= \mathbb{P}(Y = 8) = 5/36 \\ \mathbb{P}(Y = 4) &= \mathbb{P}(Y = 10) = 3/36 & \mathbb{P}(Y = 7) &= 6/36 \end{aligned}$$

In this case, the expectation for Y is 7:

$$\mathbb{E}(Y) = \sum_{i \in I(Y)} i \mathbb{P}(Y = i) = (2 + 12) \cdot \frac{1}{36} + (3 + 11) \cdot \frac{2}{36} + (4 + 10) \cdot \frac{3}{36} + (5 + 9) \cdot \frac{4}{36} + (6 + 8) \cdot \frac{5}{36} + 7 \cdot \frac{6}{36}$$

□

Under the assumption that the two events are independent from each other, we can denote the probability of two events happening as the product of the two associated probabilities:

$$\mathbb{P}(E_1 \wedge E_2) = \mathbb{P}(E_1) \mathbb{P}(E_2)$$

In probability theory, $\mathbb{P}(E_1 \wedge E_2)$ is also denoted as $\mathbb{P}(E_1, E_2)$.

Example 3. If $E_1 := D_1 = i$ and $E_2 := D_2 = j$ respectively and D_1 and D_2 denote two subsequent experiments of throwing an unrigged dice, then we have $\mathbb{P}(E_1 \wedge E_2) = \frac{1}{6} \cdot \frac{1}{6} = \frac{1}{36}$. If, on the other hand, E_2 is defined as $E_2 := D_1 = j$, we have that the same experiment cannot return two possible outcomes in the same time, and therefore $\mathbb{P}(E_1 \wedge E_2)$ will return $\frac{1}{6}$ only if $i = j$ and 0 otherwise³. □

³More formally, $\mathbb{P}(E_1 \wedge E_2) = \begin{cases} 1/6 & i = j \\ 0 & \text{oth.} \end{cases}$

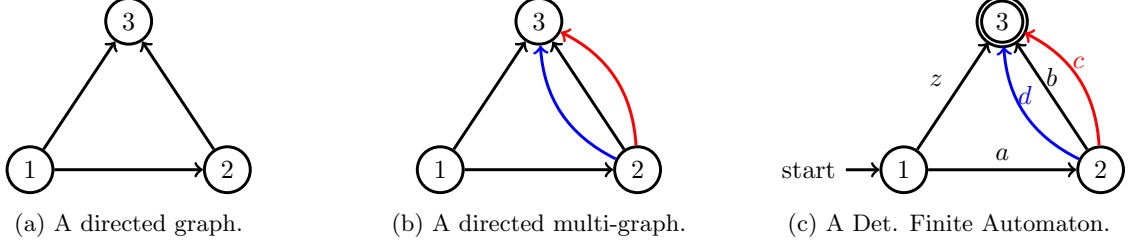


Figure 3: Three simple graphs

If no assumption is known over the events E_1 and E_2 , we have that the probability that either the first or the second event happens is defined as follows:

$$\mathbb{P}(E_1 \vee E_2) = \mathbb{P}(E_1) + \mathbb{P}(E_2) - \mathbb{P}(E_1 \wedge E_2)$$

Example 4. If $E_1 := D_1 = 1$ and $E_2 := D_2 = 4$ respectively and D_1 and D_2 denote two subsequent experiments of throwing an unrigged dice, then we have $\mathbb{P}(E_1 \wedge E_2) = \frac{6}{36} + \frac{6}{36} - \frac{1}{36} = \frac{11}{36}$. The intuition behind it is that the first experiment with probability $\mathbb{P}(E_1) = \frac{6}{36}$ will consider the configurations $\{(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6)\}$ and the second will consider the configurations $\{(1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (6, 4)\}$; therefore, $\frac{6}{36} + \frac{6}{36}$ will consider the pair (1, 4) twice and, therefore, should be removed one of these pairs, thus obtaining all the possible 11 configurations:

$$\{(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 4), (3, 4), (4, 4), (5, 4), (6, 4)\}$$

If, on the other hand, E_2 is defined as $E_2 := D_1 = 4$, we have that the same experiment cannot return two possible outcomes in the same time, and therefore $\mathbb{P}(E_1 \wedge E_2)$ will return. In this case, $\mathbb{P}(E_1 \vee E_2)$ reduces to $\mathbb{P}(E_1) + \mathbb{P}(E_2) = \frac{2}{6}$, and this corresponds to the intuition that there 1 and 4 are two possible outcomes of the experiment over a total of 6 possible outcomes. \square

2.2 Graph Theory for board modelling

A **(directed) graph** G is simply defined as a pair of sets (V, E) , where E is a subset of all the possible pairs of vertices, i.e., $E \subseteq V^2 = V \times V$. A directed graph could be represented as a square matrix M of size $|V| \times |V|$, where $M_{ij} = 1$ if there is an edge connecting vertex i with j , and 0 otherwise⁴. If an edge (i, j) is associated to a transition probability p_{ij} of transitioning to j while being in i , then we will have $M_{ij} = p_{ij}$. This specific representation will be used in the forthcoming tutorial.

As a consequence of the former definition, graph do not allow multiple edges connecting two given pair of nodes. With respect to our game of interest, let us try to model each room as a single vertex of the graph, and each door as an edge, leading from a room to the other: as we might observe, the vertices describe the previously mentioned state space, while the set of edges describe the transition model. Nevertheless, the former definition only allows at most one edge among each pair of vertices. We need to extend the previous definition.

In order to do so, the edge collection should be represented as a **multi-set**; a multiset is a pair (M, μ) , where μ is a function mapping each element of M to a non-zero natural number representing the number of instances of a same itemset (i.e., multiplicity). By doing so, we can say that any multiset (M, μ) is a sub-multiset of (M', μ') if M' contains all the items from M and the multiplicity of the elements of M is lesser or equal to the one of the corresponding items in M' .⁵; we can also specifically target the i -th instance of an item $n \in M$ as i_n , and we would state that the i_n is contained in the multi-set M if there exist at least n instances of i in M .⁶ Therefore, we call **(directed) multi-graph**⁷ G a pair $(V, (E, \mu))$, where (E, μ) is a multi-set and $E \subseteq V^2$.

⁴More formally, $M_{ij} = \begin{cases} 1 & i \rightarrow j \in E \\ 0 & \text{oth.} \end{cases}$

⁵More formally, $(M, \mu) \subseteq (M', \mu') \Leftrightarrow \forall x \in M. x \in M' \wedge \mu(x) \leq \mu'(x)$.

⁶More formally, $i_n \in (M, \mu) \Leftrightarrow i \in M \wedge n \leq \mu(i)$.

⁷Please observe that both the definition of graph and multi-graph allow graphs with infinitely enumerable vertices (for graphs) and edges (for multi-sets). As we are not interested in infinitely long games that a player will not be able to finish and that might not possibly fit in any memory, we will restrict our analysis to finite graphs and finite multi-graphs.

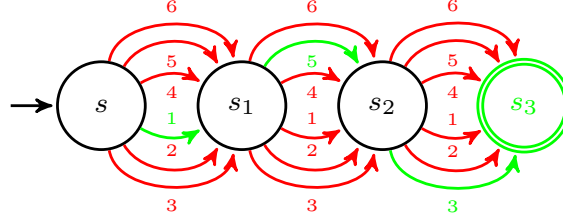


Figure 4: A multi-graph representing our game's board. Green (Red) edges represent (in)correct doors.

Example 5. Figure 3a represents a simple directed graph $G = (\{1, 2, 3\}, \{(1, 2), (1, 3), (2, 3)\})$, containing three vertices and three edges, which are represented as pair of vertex ids.

Let us now suppose to add two more edges between node 2 and 3: this will require to represent the edge sets as a multiset $M = (\{(1, 2), (1, 3), (2, 3)\}, \mu)$, where there is only one edge connecting 1 and 2 ($\mu(1, 2) = 1$), another single edge connecting 1 and 3 ($\mu(1, 3) = 1$), and three edges connecting 2 and 3 ($\mu(2, 3) = 3$). We could represent μ as the following finite function:

$$\mu := [(1, 2) \mapsto 1, (1, 3) \mapsto 1, (2, 3) \mapsto 3] \quad (1)$$

Please observe that we might represent such a finite function as a C# dictionary.

Let us assume to have an edge colouring (finite) function $\tilde{\lambda}$, mapping each edge in to a respective colour: we would have three black edges $\tilde{\lambda}((1, 2)_1) = \tilde{\lambda}((1, 3)_1) = \tilde{\lambda}((2, 3)_1) = \text{black}$, one red edge $\tilde{\lambda}((2, 3)_2) = \text{red}$, and a blue edge $\tilde{\lambda}((2, 3)_3) = \text{blue}$; we could then denote $\tilde{\lambda}$ as follows:

$$\tilde{\lambda} := [(1, 2)_1 \mapsto \text{black}, (1, 3)_1 \mapsto \text{black}, (2, 3)_1 \mapsto \text{black}, (2, 3)_2 \mapsto \text{red}, (2, 3)_3 \mapsto \text{blue}]$$

As a result, we obtain the multi-graph depicted in Figure 3b.

Given the former definition, we can now represent the board associated to our game as in Figure 4: we leave as an exercise the formal definition of such a multi-graph.

Multi-graph can be efficiently represented as adjacency lists: an adjacency list represents the set of vertices as a list, where each vertex is associated to another list describing its outgoing edges, i.e. the edges having the vertex of interest as a source. Please observe that a graph is a specific case of a multi-graph, where μ will always return 1 for each edge in E . Therefore, graphs could be also represented as adjacency lists. For our board game, we can represent such an adjacency list as a matrix `Door[][] transition_from_states`, where its first index `a ≤ total_states` represents the door id, while the second `b` represents one of the doors in `a`.

```

1 using System.Collections.Generic;
2
3 namespace ConsoleApp2.probabilities {
4     public class Board {
5         public Door[][] transition_from_states;
6         public ulong total_states;
7         public ulong total_doors;
8
9         public Board(ulong n = 4, ulong doors = 6) {
10             total_states = n;
11             total_doors = doors;
12             transition_from_states = new Door[n][];
13             for (ulong i = 0; i < n - 1; i++) {
14                 transition_from_states[i] = new Door[doors];
15                 for (ulong j = 0; j < doors; j++) {
16                     transition_from_states[i][j] = new Door();
17                 }
18             }
19             transition_from_states[n - 1] = new Door[0];
20         }
21     }
22 }

```

Now, we need to also model the Door. A door could be locked, it could be the wrong door, and could be chosen by the player in a given moment of the game. When chosen, the door will lead to a new state, which id is `reachable_state`. By default, all the doors are locked and are wrong, and therefore lead to no subsequent state.

Listing 2: probabilities/Door.cs

```

1 namespace ConsoleApp2.probabilities {
2     public class Door {
3         public bool locked;
4         public bool wrong_door;
5         public bool is_chosen { get; set; }
6         public long reachable_state;
7
8         /// <summary></summary>
9         /// <param name="locked">Whether the door is locked or not</param>
10        /// <param name="wrongDoor">Whether the door is the expected/correct one for the solution</param>
11        /// <param name="reachableState">If the door is not locked, this determines the leading state</param>
12        public Door(bool locked = true, bool wrongDoor = true, long reachableState = -1) {
13            this.locked = locked;
14            wrong_door = wrongDoor;
15            is_chosen = false;
16            reachable_state = reachableState;
17        }
18    }
19 }

```

At this stage, we are also interested in knowing which is the initial node i from which we might possible **start** the visit of the graph, and a set of **final** nodes $F \subseteq V$ denoting the places which, when reached, denote that the user followed a right/expected path within the graph.

This notion will be useful for navigating the board: given that in our use case example we have only one initial and one final node, we can then extend the previous definition of the Board class with the `initial_vertex` and the `final_vertex` fields, so that the initial state is always 0 and the final state is always the room with the highest id. Please observe that the initial state will correspond to the agent being in the `initial_vertex` s when the game has yet to start (board initialization, as in Listing 3 and lines 14-17), while the goal test is “reaching the `final_vertex` s_3 when only the correct doors are chosen after the last game run starting from s ”.

Listing 3: probabilities/BasicScenarios.cs

```

1 namespace ConsoleApp2.probabilities {
2     public abstract class BasicScenarios {
3         internal ulong nStates { get; set; }
4         internal ulong nDoors { get; set; }
5         internal SimpleNavigationFunction f;
6
7         public BasicScenarios(ulong nStates = 4, ulong nDoors = 6 ){
8             this.nStates = nStates;
9             this.nDoors = nDoors;
10            this.f = null;
11        }
12
13        /// <summary>This function returns an initialized board, b </summary>
14        public abstract Board initScenario();
15
16        /// <summary>This will reset and initialize the navigation criteria, f </summary>
17        public abstract void initSimpleNavigationFunction();
18
19        public ulong runBestCaseScenario() {
20            Board b = initScenario();
21            initSimpleNavigationFunction();
22            return SimpleNavigation.navigate(ref b, false, f);
23        }
24        public ulong runWorstCaseScenario() {
25            Board b = initScenario();
26            initSimpleNavigationFunction();
27            return SimpleNavigation.navigate(ref b, true, f);
28        }
29    }
30 }

```

This requires the game to have an additional field, `state_navigation`, which will keep track whether the player did a correct move in the i -th state of the game. By decoupling the automata's possible actions on the world (*multi-graph*: `transition_from_states` with `initial_vertex` and `final_vertex`) with the choices of the player throughout the game (`state_navigation`), we will be able to reduce the amount of information required to implement our solution. In particular, for each visited state in the board, we will add a new i -th `bool` value into `state_navigation` identifying the correctness of the chosen door in the i -th state.

```

1  using System.Collections.Generic;
2
3  namespace ConsoleApp2.probabilities {
4      public class Board {
5          // After the previously defined fields, add these other ones:
6          public List<bool> state_navigation;
7          public ulong initial_vertex;
8          public ulong final_vertex;
9
10         public Board(ulong n = 4, ulong doors = 6) {
11             // Lines to be inserted immediately before the for loop
12             initial_vertex = 0;
13             final_vertex = (n == 0 ? 0 : n - 1);
14             state_navigation = new List<bool>();
15             // End of the insertion!
16         }
17     }
18 }

```

Now we might ask ourselves: how could we possibly extend the previously defined formal model so to describe the configuration associated to a single door? In order to do so, we could “decorate” the definition of a (multi-)graph by adding (finite) functions mapping the existing nodes and/or edges to boolean values.

Edge labels are frequently exploited to identify the actions allowing the transition from one state to the other. Edges are labelled via a finite function $\lambda : E \rightarrow \Sigma^*$, where Σ^* represents the set of all the possible strings⁸. A graph “decorated” with an alphabet Σ , an edge labelling function λ , an initial node i , and a set of accepting nodes F , is usually referred as a **Finite Automata** $(V, (E, \mu), \Sigma, i, F, \lambda)$. Among all the possible automata, we are interested to the **deterministic** ones having one single edge label per outgoing edge⁹.

Example 6. Figure 3c represents a Deterministic Finite Automaton. The visit of the graph should always start from the initial node $i = 1$, and will always end in the single accepting state 3, thus $F = \{3\}$. This graph represents the regex $(a?(d|b|c))^+z$, as this graph accepts the strings d , b , c , ad , ab , ac , and z . The full DFA can be represented as the following tuple:

$$(\{1, 2, 3\}, (\{(1, 2), (1, 3), (2, 3)\}, \mu), \{a, b, c, d, z\}, \lambda, 1, \{3\})$$

where μ is defined as in Equation 1 on page 6, and the edge labelling function is defined as follows:

$$\lambda := [(1, 2)_1 \mapsto a, (1, 3)_1 \mapsto z, (2, 3)_1 \mapsto b, (2, 3)_2 \mapsto c, (2, 3)_3 \mapsto d]$$

□

In any game, these labels correspond to the actions that the agent must perform in order to change the state he is in where, when available, the state is defined by the combination of the current visited node jointly with other world information (e.g., visited doors). In our scenario, the element distinguishing each possible door is its id. Therefore, we have that, for the i -th door connecting s to s' , we have $\lambda((s, s')_i) = i$ and `transition_from_states[s][i].reachable_state == s'`. In the next section, we are going to denote an edge connecting s to s' with an arrow, which is also labelled as the label in λ (e.g., $s \xrightarrow{\lambda((s, s')_i)} s'$).

⁸This notation comes from the set theory exploited for regular languages, where Σ is an **alphabet**, Σ^n denotes the set of all the possible strings of length n on the alphabet Σ , and Σ^* is defined as the union over all the possible Σ^n for each $n \in \mathbb{N}$; more formally, $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$.

⁹More formally, $\forall u \in V. \forall a \in \Sigma. \exists! v \in V. \exists! i \leq \mu((u, v)). \lambda((u, v)_i) = a$.

3 Episodically vs. Sequentially Visiting

As we have seen during the lecture, by changing the game's rules we could have very different scenarios. The scenarios that you encountered in today's lecture are the following:

1. For each state, we test the doors sequentially; all the incorrect doors are closed (but this is not known by the player until they open the door).
2. **Without strategy:** for each state, we pick **one door**; all the incorrect doors are **left open**. **If we reach s_3 through at least one incorrect door (i.e., fail), we backtrack to s and keep the same three correct doors.**
3. **With strategy:** (as #2).
4. For each state, we pick one door. If we reach s_3 through at least one incorrect door (i.e., **fail**), we backtrack to s and **randomly select another triplet of correct doors**.
5. For each state, we pick one door **after playing the Monty Hall Game**. If we reach s_3 through at least one incorrect door (i.e., **fail**), we backtrack to s and randomly select another triplet of correct doors.

In order to remark the common parts that are shared which are the main differences within episodic games, we are going to use C# inheritance and overloading mechanisms for implementing different possible behaviours. On the other hand, for the third strategy we will focus on an *ad-hoc* algorithm for efficiently solving the game and encoding a strategy from scratch. For the last two strategies, we upgrade the notion of a `Board[][]` matrix so run a single instance of the Monty Hall Problem.

3.1 Episodical Board visiting (Use Case #1 and #2)

The main difference between the first and the second use case are related to the board initialization (e.g., whether some doors might be locked or not) and the way to visit the board itself (e.g., whether the player has to backtrack to the initial state and room after a detected failure or not). If we need to do some backtracking, we would need to re-initialize the board (Listing 6, line 64). For the former we would require to change the way the board is initialized (Listing 3, lines 14-17), while for the latter we would need to also change the way we traverse the graph (Listing 3, line 5). As remarked in the interface `SimpleNavigationFunction`, the main differences between these scenarios are related to the environment's behaviour while attempting to open a wrong door (Listing 5, line 5).

Listing 4: probabilities/SimpleNavigation.cs (part 1)

```
1 using System;
2
3 namespace ConsoleApp2.probabilities {
4     public interface SimpleNavigationFunction {
5         bool onWrongDoorDo(ref ulong x, ref Board b, ref Door d);
6         void onWrongStateDo(ref ulong currState, ref Board b);
7     }
8
9     //<definition of the SimpleNavigation class>
10 }
```

For understanding how to actually implement the abstract methods and the classes from the interface, we need to sketch first an “episodic” way to visit the graph. When the player selects a given door as a correct door, we need to call the method `current_door_step` in order to assess whether the action “open the door” will lead to a transition or not. If we reach a correct door, then the test in Listing 5 at line 47 fails, and therefore we execute the **else** condition. Therefore, we will progress towards the state indicated by the correct door (line 29) and remember that the currently visited room was correct (line 30); as we will see later on, we also need to return **true** so to break the scan over the remaining possible doors.

For both use cases, we will always assume that the winning doors will be the last door for each room. Therefore, we could easily mimic the *best-case scenario* of always picking the right door by always choosing the last door first (lines 51-55).

On the other hand, the difference between the first and second use case is the following: for the first, the wrong doors are always locked (line 22), while for the second they can be chosen (the previous line is never run), thus leading to a door choice that does not satisfy the goal condition (lines 24-26). In order to mimic the *worst-case scenario*, the first one can be achieved by scanning all the doors from the first to the last (lines 47-51), while the second can be obtained by always selecting the first door. This last condition requires to kill the iteration when a wrong door is encountered: when we visit a door for the current room, we will need to determine whether we have to halt (i.e., **skip**) the iteration or not (lines 49 and 53). This also implies that, the method invoked at line 26 should return **true**.

Listing 5: probabilities/SimpleNavigation.cs (part 2)

```

1  //<definition of the SimpleNavigation class>
2  class SimpleNavigation {
3      /// <summary>
4      /// Performs the game-play navigation towards the next step. It uses a SimpleNavigationFunction to
5      /// ↪ determine the best policy to adopt
6      /// </summary>
7      /// <param name="board">Board on which we are playing</param>
8      /// <param name="curr_state">Current state within the board</param>
9      /// <param name="d">Board currently chosen from the game</param>
10     /// <param name="door_id">Id associated to the current door</param>
11     /// <param name="attempts">Counting the number of attempts</param>
12     /// <param name="set_chosen">Whether the same door was chosen in a previous iteration of the game</
13     /// ↪ param>
14     /// <param name="f">Handling function for </param>
15     /// <returns>Returns true if you need to kill the iteration over the doors to visit, and false
16     /// ↪ otherwise</returns>
17     public static bool current_door_step(ref Board board,
18                                         ref ulong curr_state,
19                                         long door_id,
20                                         ref ulong attempts,
21                                         SimpleNavigationFunction f) {
22         ref Door d = ref board.transition_from_states[curr_state][door_id];
23
24         attempts++;
25         if (d.locked)
26             Console.WriteLine("Door is locked!");
27         else if (d.wrong_door) {
28             Console.WriteLine("Wrong Door!");
29             return f.onWrongDoorDo(ref curr_state, ref board, ref d);
30         } else {
31             Console.WriteLine("Moving towards state_{0}", d.reachable_state);
32             curr_state = (ulong)d.reachable_state; // Moving towards the next state as indicated by the
33             ↪ correct door
34             board.state_navigation.Add(true); // Set the current door as correct
35             return true; // Break the iteration!
36         }
37         return false; // Do not break the iteration
38     }
39
40     public static ulong navigate(ref Board board,
41                                 bool worst_case_scenario,
42                                 SimpleNavigationFunction f) {
43         ulong attempts = 0;
44         ulong curr_state = board.initial_vertex;
45         if (worst_case_scenario) Console.WriteLine("Simulating the Worst Case Scenario");
46         else Console.WriteLine("Simulating the Best Case Scenario");
47         while (curr_state != board.final_vertex) {
48             Console.WriteLine("Current State_{0}", curr_state);
49             int doorNo = board.transition_from_states[curr_state].Length;
50             if (worst_case_scenario) {
51                 for (int doorId = 0; doorId < doorNo; doorId++) {
52                     if (current_door_step(ref board, ref curr_state, doorId, ref attempts, f)) break;
53                 }
54             } else {
55                 for (int doorId = doorNo-1; doorId >= 0; doorId--) {
56                     if (current_door_step(ref board, ref curr_state, doorId, ref attempts, f)) break;
57                 }
58             }
59             if (curr_state == board.final_vertex) {
60                 bool is_ok = true; // Break the iteration only if there is no wrong door
61             }
62         }
63     }
64 }

```

```

58         foreach (bool v in board.state_navigation) { // Checking if in each room we visited the correct
59             ↪ door
60             if (!v) {
61                 is_ok = false; // if not, invoke onWrongStateDo
62                 break;
63             }
64             if (!is_ok) f.onWrongStateDo(ref curr_state, ref board);
65         }
66     }
67     Console.WriteLine("Total_attempts={0}", attempts);
68     return attempts;
69 }
70 }

```

As a last word, we are also counting the number of total attempts at opening a door: the total number of attempts of opening a door `attempts` is incremented in `current_door_step` but, given that this variable is declared in `navigate` and passed by reference (`ref`) to `current_door_step`, the correctly obtained value is correctly printed (line 67) and returned (line 68). Similar considerations might also be applied to the `curr_state`: as it is always passed by reference (e.g., line 64), we can easily update the current room if we need to backtrack to the initial room.

Now, we are ready to set implement both the first and the second scenario. The solution for the first implementation is provided in Listing 6. As all the wrong doors will be also be locked, neither `onWrongDoorDo` nor `onWrongStateDo` will be invoked: we extend `SimpleNavigationFunction` into a `OnWrongDoNothing` which will never change the game execution. This behaviour is then injected into the base class via the method `initSimpleNavigationFunction` (label 29). With respect to the board initialization, let us remember that each `Door` is set as wrong and locked by default: therefore, for this scenario we will only need to change the last door (line 18) by unlocking it (line 20) and by setting it as correct (line 21); last, each door in the s -th door (counting from zero) will lead to the $(s + 1)$ -th door (line 22).

Listing 6: probabilities/FirstScenario.cs

```

1 namespace ConsoleApp2.probabilities {
2     class FirstScenario : BasicScenarios {
3         class OnWrongDoNothing : SimpleNavigationFunction {
4             // As all the wrong doors are locked and whether the door is locked or not is always tested as a
5             // first condition, this method will be never run! Returning a possible boolean value.
6             public bool onWrongDoorDo(ref ulong x, ref Board b, ref Door d) { return true; }
7             // This method will not be run either: so, perform no operation
8             public void onWrongStateDo(ref ulong currState, ref Board b) { /* noop */ }
9         }
10
11         public FirstScenario(ulong n = 4, ulong doors = 6) : base(n, doors) { }
12
13         public override Board initScenario() {
14             Board board = new Board(nStates, nDoors);
15             long s = 0;
16             foreach (Door[] adj in board.transition_from_states) {
17                 if (adj.Length > 0) {
18                     ref Door x = ref adj[adj.Length - 1];
19                     x.locked = false;
20
21                     x.wrong_door = false;
22                     x.reachable_state = ++s;
23                 }
24             }
25             return board;
26         }
27
28         public override void initSimpleNavigationFunction() {
29             if (f == null) f = new OnWrongDoNothing();
30         }
31     }
32 }

```

The solution for the second scenario is provided in Listing 7. In this other scenario, when we reach a wrong door, we should still be able to progress towards the next door (line 15), but we should remember that we made an incorrect choice that will lead into a non acceptance state: therefore, we should add `false` into the `state_navigation` (line 16). Furthermore, the currently chosen door should be picked, and therefore the iteration among the remaining doors should be stopped (line 17). Furthermore, for resetting the state of the Board and backtracking to the initial position it is sufficient to both set the current room to the initial room (line 26) and to clear the `state_navigation` list. For the Board initialization we shall scan over all the possible doors, as no door should be locked (line 41), all the doors in room i should lead to the next room $i + 1$ (line 42), and only the last door should be set as correct (line 43)

Listing 7: probabilities/SecondScenario.cs

```

1 using System;
2
3 namespace ConsoleApp2.proBABILITIES {
4
5     public class OnWrongDoBacktrack : SimpleNavigationFunction {
6         /// <summary>
7         /// When a wrong door is choosen, navigates towards a state while remembering that the state was
8         ///     ↪ incorrect
9         /// </summary>
10        /// <param name="currState">Current state from which start the navigation</param>
11        /// <param name="b">Board containing the stateful information</param>
12        /// <param name="d">Door that was currently choosen</param>
13        /// <returns></returns>
14        public bool onWrongDoorDo(ref ulong currState, ref Board b, ref Door d) {
15            Console.WriteLine("___Moving_towards_state_{0}", d.reachable_state);
16            currState = (ulong)d.reachable_state;
17            b.state_navigation.Add(false); // Add a wrong state to the navigation
18            return true; // Continue playing the game without interrupting the game
19        }
20
21        /// <summary>
22        /// Backtracks the game play to the initial state of the game
23        /// </summary>
24        /// <param name="currState">Current state from which perform the backtrack</param>
25        /// <param name="b">Boad containing the initial state information from which re-start the game</param>
26        public void onWrongStateDo(ref ulong currState, ref Board b) {
27            currState = b.initial_vertex;
28            b.state_navigation.Clear();
29        }
30    }
31
32    public class SecondScenario : BasicScenarios {
33        public SecondScenario(ulong n = 4, ulong doors = 6) : base(n, doors) { }
34
35        public override Board initScenario() {
36            Board board = new Board(nStates, nDoors);
37            long i = 0, M = board.transition_from_states.Length; // Counting the number of the possible states
38            foreach (Door[] adj in board.transition_from_states) {
39                long j = 0; // Current door id, counting from zero
40                long N = (long)adj.Length; // Maximum door id
41                foreach (Door refD in adj) {
42                    refD.locked = false; // No door should be locked
43                    refD.reachable_state = (i == (M-1) ? -1 : i+1);
44                    if (j == (N - 1)) refD.wrong_door = false;
45                    j++;
46                }
47                i++;
48            }
49            return board;
50        }
51
52        public override void initSimpleNavigationFunction() {
53            if (f == null) f = new OnWrongDoBacktrack();
54        }
55    }

```

3.2 An time-efficient algorithm for Randomic Guessing (Use Case #3)

Let us remember that, in this scenario, the set of the three winning doors is never changed. As we observed in today's lecture, the worst case scenario for a player is to continuously choose a wrong configuration, possibly always the same one. This is what it happens if the encoded strategy is wrong and the player stubbornly sticks to his original decision.

Nevertheless, we would like to adopt a strategy that, in the worst case scenario, will visit all of the possible 6^3 combinations of doors. A straightforward solution would require to either iterate and test over all the possible 6^3 winning door configurations. On the other hand, this solution will require very little memory, as we will only need three variables for each door configuration. This solution's worst case scenario is still more efficient than generating three random numbers between 0 and 5 ($i = \mathcal{U}(0, 5)$, $j = \mathcal{U}(0, 5)$, $k = \mathcal{U}(0, 5)$), discarding the previously generated configurations that were not the winning ones, and testing the newly generated configurations. Still, in average, the probability of picking a wrong door will exponentially decrease as we increase the number of possible attempts in finding a possible solution. An example of this solution is given in the following code:

```
1 using ConsoleApp2.utils;
2 using System;
3 using System.Collections.Generic;
4 using System.Diagnostics;
5 using System.Linq;
6
7 namespace ConsoleApp2.probabilities
8 {
9     /// <summary>
10     /// The Third scenario differs from the second by just the implementation of a simple strategy (the choice
11     /// ↔ of remembering the previous choice)
12     /// </summary>
13     public class ThirdScenario {
14         Random generator_s1;
15         Random generator_s2;
16         Random generator_s3;
17         HashSet<utils.Pair<UInt64, utils.Pair<UInt64, UInt64>>> s1_s2_s3_attempts;
18         utils.UniformRandom door_distr;
19         Board default_board;
20         SecondScenario sn;
21         int doors;
22
23         public ThirdScenario(int seed) {
24             sn = new SecondScenario(4, 6);
25             door_distr = new utils.UniformRandom(0, 5); // Transformation  $\mathcal{U}(0, 1) \mapsto \mathcal{U}(0, \text{doors} - 1)$ 
26             generator_s1 = new Random(seed+0); // The input seed is associated to the first door generator
27             generator_s2 = new Random(seed+1); // The remaining generators will have different seeds
28             generator_s3 = new Random(seed+2);
29             this.doors = (int)6;
30         }
31
32         void setSeed(int seed) {
33             generator_s1 = generator_s2 = generator_s3 = null;
34             generator_s1 = new Random(seed + 0);
35             generator_s2 = new Random(seed + 1);
36             generator_s3 = new Random(seed + 2);
37         }
38
39         void clearAll() {
40             if (s1_s2_s3_attempts != null)
41                 s1_s2_s3_attempts.Clear();
42             else
43                 s1_s2_s3_attempts = new HashSet<utils.Pair<UInt64, utils.Pair<UInt64, UInt64>>>();
44         }
45
46         private static utils.Pair<UInt64, UInt64> genPair(int i, int j) {
47             return new utils.Pair<UInt64, UInt64>((uint)i, (uint)j);
48         }
49
50         private static utils.Pair<UInt64, utils.Pair<UInt64, UInt64>> genTriple(int i, int j, int k) {
51             return new utils.Pair<UInt64, utils.Pair<UInt64, UInt64>>((uint)i, new utils.Pair<UInt64, UInt64>((
52                 ↔ uint)j, (uint)k));
```

```

53 public ulong testRandomCaseScenario(bool debugInfo = false) {
54     default_board = sn.initScenario(); // Using the same board generated by the second scenario
55     clearAll(); // Re-initializing the data structures used for guessing the internal value
56     // Counting the attempts
57     ulong attempts = 0;
58
59     bool not_found = true;
60     while (not_found) { // Continue to loop until the winning configuration is not found
61         int i, j, k;
62         i = door_distr.nextInt(ref generator_s1); // Generating the first room's door
63         j = door_distr.nextInt(ref generator_s2); // Generating the second room's door
64         k = door_distr.nextInt(ref generator_s3); // Generating the third room's door
65
66         // If we have already inserted the tuple in s1_s2_s3_attempts, then it means that we have already
67         // recognized it as a wrong configuration. So, we don't have to check the board.
68         bool alreadyMet = s1_s2_s3_attempts.Contains(genTriple(i, j, k));
69
70         if (alreadyMet ||
71             // "If the randomly generated alternative is the wrong one" means
72             // "If there exists at least one door which is wrong."
73             default_board.transition_from_states[0].ElementAt((int)i).wrong_door ||
74             default_board.transition_from_states[1].ElementAt((int)j).wrong_door ||
75             default_board.transition_from_states[2].ElementAt((int)k).wrong_door) {
76             if (debugInfo) Console.WriteLine("Wrong_configuration:_{0}_{1}_{2}!", i, j, k);
77             if (!alreadyMet) s1_s2_s3_attempts.Add(genTriple(i, j, k)); // Adding the triple only if we
78                                 //haven't previously added it.
79         } else {
80             if (debugInfo) Console.WriteLine("Winning_configuration:_{0}_{1}_{2}!", i, j, k);
81             not_found = false;
82         }
83         attempts++;
84     }
85
86     if (debugInfo) {
87         Console.WriteLine("Attempts:_{0}", attempts+1);
88         Console.WriteLine("");
89         Console.WriteLine("");
90     }
91     Debug.Assert(attempts <= 216, "There_should_be_at_most_216_attempts:_we_got_" + (attempts).ToString
92                  ↪ ());
93     return (attempts + 1);
94 }
95 }
96 }

```

The constructor as of this class accepts as a sole parameter the seed that is going to be associated to the three distinct random generators associated to each of the room of the game containing a door: `generator_s1` for room s , `generator_s2` for room s_1 , and `generator_s3` for room s_2 . The seed of such generators can be also changed at runtime by invoking the `setSeed` method: this will be extremely useful in the test phase, where we are going to assess the distribution of the number of attempts via the minimum, maximum, and average number of attempts by varying the value of the seed.

As we will observe after a short digression on the C# language, we could still upgrade this code so to generate an algorithm which has a computational complexity which avoids to test multiple times randomly-generated configurations that were previously discarded. In order to reduce the number of randomly-generated triples that needs to be tested, we might avoid generating an i or j value that might have been discarded given the previous observations.

This simple code allows us to deepen some interesting aspects of the C# programming language: in fact, we might ask ourselves how to avoid inserting multiple equivalent Pairs into the `HashSet s1_s2_s3_attempts`. In order to do so, the class should provide an equality binary relation as well as a hashing function. The motivation behind the previous assertion is the following: in order to reduce the computation time of performing, the `HashSet` is implemented over a *hash table*, where each element x having a hash value of $h(x)$ is placed in its $h(x)$ -th slot, also referred as *bucket*. For inserting x into a `HashSet`, such a class will firstly test if there does not exist a bucket with id $h(x)$: if not, a new slot is initialized and the element is inserted inside the newly created slot; otherwise, the `HashSet` will insert x in the bucket having index $h(x)$ if and only if such a bucket does not contain another element which is equivalent to x . After observing that two pairs (i, j) and (h, k) are equivalent only if i is h

and j is k (line 15)¹⁰, and that a possible hashing value for (i, j) is defined by xor-ing the binary digits of the hash value associated to both i and j by computing $h(i) \oplus h(j)$ (line 23)¹¹, we can implement `Pair` as follows:

Listing 8: `utils/Pair.cs`

```

1 using System;
2 using System.Collections.Generic;
3
4 namespace ConsoleApp2.utils {
5     public class Pair<TKey, TValue> : IEquatable<Pair<TKey, TValue>> {
6         public TKey key; // first element of the pair
7         public TValue value; // second element of the pair
8
9         public Pair(TKey key, TValue value) {
10             this.key = key;
11             this.value = value;
12         }
13
14         public bool Equals(Pair<TKey, TValue> y) {
15             return key.Equals(y.key) && value.Equals(y.value);
16         }
17
18         public override bool Equals(object obj) {
19             return obj is Pair<TKey, TValue> && Equals((Pair<TKey, TValue>)obj);
20         }
21
22         public override int GetHashCode() {
23             return key.GetHashCode() ^ value.GetHashCode(); // Or something like that
24         }
25     }
26 }

```

Given this definition, we can conveniently assume that a triple (i, j, k) is encoded as a pair $(i, (j, k))$, where (j, k) is yet another pair.

In order to reduce the possible number of numbers to be tested and/or randomly generated, we can do the following considerations:

1. As previously stated, if I have already tested the door configuration (i, j, k) and I have checked that this configuration is wrong, then on the next attempts I should always discard door configurations that are equivalent to (i, j, k) . This condition was already tested through `s1.s2.s3_attempts`.
2. Given the doors i and j respectively associated to the first and second room, we could discard any possible configuration (i, j, \bar{k}) for any \bar{k} if we already discarded 6 different configurations starting with (i, j, \dots) , which are $\{(i, j, 1), (i, j, 2), \dots, (i, j, 6)\}$.

Similar considerations could be also done for the following configurations, where i and j are fixed and \bar{k} varies: (i, \bar{k}, j) and (\bar{k}, i, j) .

These conditions could be tested via an object of type `Dictionary<utils.Pair<UInt64, UInt64>, HashSet<UInt64>>`, where the dictionary's keys reflect the pairs (i, j) from the previous configurations, and the values associated to the keys reflect the previously discarded k values for the remaining room. E.g., an object `s1.s3.position_to_other_choices` recognizes (i, k, j) as a previously discarded solution if there exists a key (i, j) associated to a `HashSet` value containing the number k ; furthermore, `s1.s3.position_to_other_choices` will discard any configuration (i, \bar{k}, j) if the key (i, j) maps to a `HashSet` value containing at least 6 elements.

3. Given the door i associated to the first room, we could discard any possible configuration (i, \bar{j}, \bar{k}) for any \bar{j} and \bar{k} associated respectively to the second and third door if we already discarded 36 different configurations starting with i , which are $\{(i, 1, 1), (i, 1, 2), \dots, (i, 1, 6), (i, 2, 1), (i, 2, 2), \dots, (i, 6, 6)\}$.

Similar considerations could be also done for the following configurations: (\bar{k}, i, \bar{j}) and (\bar{k}, \bar{j}, i) .

These conditions could be tested via an array `HashSet<utils.Pair<UInt64, UInt64>>[]`, where the index of the array refers to the door-id associated to a given room, and the content of the `HashSet` reflect the previously discarded pairs for the other rooms. E.g., an object

¹⁰More formally. $\forall i, j, h, k. (i, j) = (h, k) \Leftrightarrow i = h \wedge j = k$

¹¹More formally, $h((i, j)) = h(i) \oplus h(j)$

`s2_position_to_other_choices` recognizes (j, i, k) as a previously discarded solution if the i -th `HashSet` of the array (counting from zero) contains a pair (j, k) ; furthermore, `s2_position_to_other_choices` will discard any configuration (\bar{j}, i, \bar{k}) if the i -th `HashSet` of the array (counting from zero) contains 36 elements.

Nevertheless, this solution will require, in the worst case scenario, more memory than the naïve solution where we generate and store all the possibly winning door configuration, and then test each one of them. Nevertheless, this is the price to pay for mimicking a naïve inference process for both generating and accepting pseudo-randomly generated door triplets that have not previously met. While implementing this proposed solution, you might consider the following suggested shortcuts:

- An array of `HashSets` of length n could be initialized by exploiting C#'s `Ranges` and `Linq's Select`: in particular `Enumerable.Range(0, n)` will generate an iterable object returning all the elements from 0 to $n - 1$ included, while each of these numbers could be mapped into a new `HashSet` via the `Select` operator; last, the resulting iterable object of `HashSets` can be converted into an array by invoking the `ToArray()` method. In particular, `s2_position_to_other_choices` might be conveniently initialized in the `clearAll` method as follows (Listing 9, line 38):

```
1 s2_position_to_other_choices = Enumerable.Range(0, (int)doors).Select(i => new HashSet<utils.Pair<
    ↪ UInt64, UInt64>>()).ToArray();
```

- The `GetOrInsert` method declared in the `utils/CppStructUtils.cs` file allows to mimic the behaviour of `operator[]` from C++'s `std::map` and `std::unordered_map`, where no exception is ever raised if the key is missing from the map. This means that, given a `Dictionary<K,V>` object `x`, `x.GetOrInsert(key, f)` returns a reference to its mapped value if `key` is a valid dictionary key. Otherwise, the function inserts a new element with that key generated via `f` and returns a reference to its mapped value. Such methods are implemented as follows:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4
5 namespace ConsoleApp2.utils {
6     // <class EqualityHashSet>
7
8     public static class CppStructUtils {
9
10         // <definition of IsSubsetOf>
11
12         /// <summary>
13         /// This function mimicks the operator[] for maps, where if the key is not in the map, then the
14         ↪ map
15         /// generates a new default value and associates it to the key, and then returns the newly
16         ↪ created value,
17         /// and otherwise returns the value associated in the key in the map
18         /// </summary>
19         /// <typeparam name="K"></typeparam>
20         /// <typeparam name="V"></typeparam>
21         /// <param name="d"></param>
22         /// <param name="toFind"></param>
23         /// <param name="generator"></param>
24         /// <returns></returns>
25         public static V GetOrInsert<K, V>(this Dictionary<K, V> d, K toFind, Func<V> generator) {
26             if (d.ContainsKey(toFind))
27                 return d[toFind];
28             else {
29                 var x = generator();
30                 d.Add(toFind, x);
31                 return x;
32             }
33         }
34
35         public static HashSet<UInt64> GetOrInsert(this Dictionary<utils.Pair<UInt64, UInt64>, HashSet<
36             ↪ UInt64>> d, utils.Pair<UInt64, UInt64> toFind) {
37             if (d.ContainsKey(toFind))
38                 return d[toFind];
39             else {
40                 var x = new HashSet<UInt64>();
```



```

38         d.Add(toFind, x);
39         return d[toFind];
40     }
41 }
42
43 // <definition for Emplace>
44 }
45 }

```

As a final hint for the implementation of the algorithm, all the objects' initialization, such as the objects of type `Dictionary<utils.Pair<UInt64, UInt64>, HashSet<UInt64>>`. The solution to the given exercise is given in the following Listing; we will have the following conditions:

1. While generating the first door configuration, i , we can discard a randomly-generated value for it (line 81) only if we know that this door failed with other 36 configurations from the next rooms (line 84). Therefore, we need to continue to generate numbers until we find a first door configuration that we haven't previously exhaustively explored.
2. While generating the second door configuration, j , we can discard the randomly-generated value for it (line 89) with a similar consideration as for i (line 92). In addition to this, we know that we can discard a door configuration (i, j, \bar{k}) for any \bar{k} if and only if we tested all the possible 6 configuration having doors i and j respectively in the first and second room (line 93).
3. Similar considerations might be done for the third door configuration, k .
4. Last, fill in all the relevant data structures when we discover a novel wrong configuration (lines 114-122).

Listing 9: probabilities/ThirdScenario.cs

```

1 using ConsoleApp2.utils;
2 using System;
3 using System.Collections.Generic;
4 using System.Diagnostics;
5 using System.Linq;
6
7 namespace ConsoleApp2.probabilities {
8     /// <summary>
9     /// The Third scenario differs from the second by just the implementation of a simple strategy (the choice
10     /// ↪ of remembering the previous choice)
11     /// </summary>
12     public class ThirdScenario {
13         Random generator_s1;
14         Random generator_s2;
15         Random generator_s3;
16         HashSet<utils.Pair<UInt64, UInt64>>[] s1_position_to_other_choices,
17         s2_position_to_other_choices,
18         s3_position_to_other_choices;
19         Dictionary<utils.Pair<UInt64, UInt64>, HashSet<UInt64>> s1_s2_position_to_other_choices,
20         s1_s3_position_to_other_choices,
21         s2_s3_position_to_other_choices;
22         HashSet<utils.Pair<UInt64, utils.Pair<UInt64, UInt64>>> s1_s2_s3_attempts;
23         utils.UniformRandom door_distr;
24         Board default_board;
25         SecondScenario sn;
26         int doors;
27
28         // <skipping both the constructor definition and the setSeed method>
29
30         void clearAll() {
31             if (s1_position_to_other_choices != null)
32                 foreach (var x in s1_position_to_other_choices) x.Clear();
33             else
34                 s1_position_to_other_choices = Enumerable.Range(0, (int)doors).Select(i => new HashSet<utils.Pair<
35                 ↪ UInt64, UInt64>>()).ToArray();
36
37             if (s2_position_to_other_choices != null)
38                 foreach (var x in s2_position_to_other_choices) x.Clear();
39             else

```

```

38     s2_position_to_other_choices = Enumerable.Range(0, (int)doors).Select(i => new HashSet<utils.Pair<
        ↳ UInt64, UInt64>>()).ToArray();
39
40     if (s3_position_to_other_choices != null)
41     foreach (var x in s3_position_to_other_choices) x.Clear();
42     else
43     s3_position_to_other_choices = Enumerable.Range(0, (int)doors).Select(i => new HashSet<utils.Pair<
        ↳ UInt64, UInt64>>()).ToArray();
44
45     if (s1_s2_position_to_other_choices != null)
46     s1_s2_position_to_other_choices.Clear();
47     else
48     s1_s2_position_to_other_choices = new Dictionary<utils.Pair<UInt64, UInt64>, HashSet<UInt64>>();
49
50     if (s1_s3_position_to_other_choices != null)
51     s1_s3_position_to_other_choices.Clear();
52     else
53     s1_s3_position_to_other_choices = new Dictionary<utils.Pair<UInt64, UInt64>, HashSet<UInt64>>();
54
55     if (s2_s3_position_to_other_choices != null)
56     s2_s3_position_to_other_choices.Clear();
57     else
58     s2_s3_position_to_other_choices = new Dictionary<utils.Pair<UInt64, UInt64>, HashSet<UInt64>>();
59
60     if (s1_s2_s3_attempts != null)
61     s1_s2_s3_attempts.Clear();
62     else
63     s1_s2_s3_attempts = new HashSet<utils.Pair<UInt64, utils.Pair<UInt64, UInt64>>>();
64 }
65
66 // <skipping the definitions of genPair and genTriple>
67
68 public ulong testRandomCaseScenario(bool debugInfo = false) {
69     default_board = sn.initScenario();
70     clearAll();
71     // Counting the attempts
72     ulong attempts = 0;
73
74     bool not_found = true;
75     while (not_found)
76     {
77         int i, j, k;
78         ulong tryouts = 0;
79         do {
80             tryouts++;
81             i = door_distr.nextInt(ref generator_s1);
82             if (debugInfo && (tryouts == 1)) Console.WriteLine("i={0}", i);
83             if (debugInfo && (tryouts > 1)) Console.WriteLine("changing_i_to_{0}", i);
84         } while (s1_position_to_other_choices[i].Count == 36);
85
86         tryouts = 0;
87         do {
88             tryouts++;
89             j = door_distr.nextInt(ref generator_s2);
90             if (debugInfo && (tryouts == 1)) Console.WriteLine("i,j={0},{1}", i, j);
91             if (debugInfo && (tryouts > 1)) Console.WriteLine("i={0},_changing_j_to_{0}", i, j);
92         } while ((s2_position_to_other_choices[j].Count == 36) ||
93             (s1_s2_position_to_other_choices.GetOrInsert(genPair(i, j)).Count == 6));
94
95         tryouts = 0;
96         do {
97             tryouts++;
98             k = door_distr.nextInt(ref generator_s3);
99             if (debugInfo && (tryouts == 1)) Console.WriteLine("i,j,k={0}", i, j, k);
100             if (debugInfo && (tryouts > 1)) Console.WriteLine("i,j,k={0},{1};_changing_k_to_{0}", i, j, k);
101         } while ((s1_s2_s3_attempts.Contains(genTriple(i, j, k)) ||
102             (s3_position_to_other_choices[k].Count == 36) ||
103             (s1_s3_position_to_other_choices.GetOrInsert(genPair(i, k)).Count == 6) ||
104             (s2_s3_position_to_other_choices.GetOrInsert(genPair(j, k)).Count == 6));
105
106
107
108

```

```

109         if (default_board.transition_from_states[0].ElementAt((int)i).wrong_door ||
110             default_board.transition_from_states[1].ElementAt((int)j).wrong_door ||
111             default_board.transition_from_states[2].ElementAt((int)k).wrong_door) {
112             if (debugInfo)
113                 Console.WriteLine("Wrong_configuration:_{0}_{1}_{2}!", i, j, k);
114             s1_position_to_other_choices[i].Add(genPair(j, k));
115             s1_position_to_other_choices[j].Add(genPair(i, k));
116             s1_position_to_other_choices[k].Add(genPair(i, j));
117
118             s1_s2_position_to_other_choices.GetOrInsert(genPair(i, j)).Add((uint)k);
119             s1_s3_position_to_other_choices.GetOrInsert(genPair(i, k)).Add((uint)j);
120             s2_s3_position_to_other_choices.GetOrInsert(genPair(j, k)).Add((uint)i);
121
122             s1_s2_s3_attempts.Add(genTriple(i, j, k));
123         } else {
124             if (debugInfo)
125                 Console.WriteLine("Winning_configuration:_{0}_{1}_{2}!", i, j, k);
126             not_found = false;
127         }
128         attempts++;
129     }
130
131     if (debugInfo) {
132         Console.WriteLine("Attempts:_{0}", attempts+1);
133         Console.WriteLine("");
134         Console.WriteLine("");
135     }
136     Debug.Assert(attempts <= 216, "There_should_be_at_most_216_attempts:_we_got_" + (attempts).ToString
137         ↪ ());
138     return (attempts + 1);
139 }
140 }
141 }

```

3.3 The (Generalized) Monty Hall Problem (Use Case #4 and #5)

In this specific use case, we want to replace the way that the player with an instance of the Generalized Monty Hall game. The simple version of the problem, with three doors and only one winning door, was formulated by Marilyn Mach in 1990 as follows:

Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say №1, and the host, who knows what's behind the doors, opens another door, say №3, which has a goat. He then says to you, "Do you want to pick door №2?" Is it to your advantage to switch your choice?

Despite this text might appear clear, there are still some hidden assumptions that are given for granted, thus making the context less clear. In particular, (i) the game host must always pick a door that was not previously picked by the contestant, (ii) the host always offer the possibility of changing the door after showing one other non winning door. Furthermore, as the host has a full knowledge of the environment, it will always show a door that has a goat and never a goat. Therefore, this problem can be easily encoded in a computer program by following the subsequent unambiguous steps:

1. Behind each of the three doors, there is either a car or a goat, for a total of two goats and one car; the probability that the car is behind a given door is identical for all doors.
2. The player chooses one of the doors; its content is not revealed yet.
3. The (game) host always knows what hides behind each door.
4. The host must open one of the unselected doors, and must offer the player the possibility to change his choice.
5. The host will always open a door that hides a goat; i.e., if the player has chosen a door that hides one goat, the host will open the door that hides the other goat. If, on the other hand,

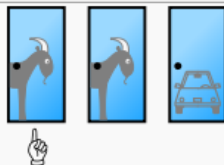
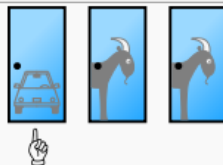
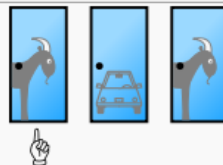
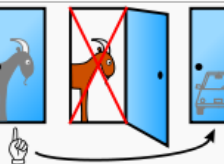
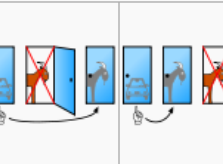
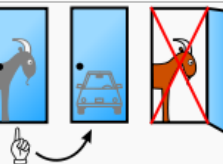
Car hidden behind Door 3	Car hidden behind Door 1	Car hidden behind Door 2
Player initially picks Door 1		
		
Host must open Door 2	Host randomly opens either goat door	Host must open Door 3
		
Probability 1/3	Probability 1/6	Probability 1/6
Switching wins	Switching loses	Switching loses
If the host has opened Door 2, switching wins twice as often as staying		If the host has opened Door 3, switching wins twice as often as staying

Figure 5: A graphical representation for the Monty Hall Problem. (Wikipedia.en)

the player has chosen the door that hides the car, the host randomly chooses one of the two remaining doors;

6. The host offers the player the option to claim what is behind the goal he originally chose, or to change, by claiming what is behind the remaining goal.

Under this assumption, we can see that the player has a $\frac{1}{3}$ chance of winning if he sticks with their original decision (as the information provided by the host does not change the player's strategy), while their probability increase to $\frac{2}{3}$ if they choose to change the door after seeing another wrong door (Figure 5):

- The player chooses the door hiding goat №1. The host shows goat №2. By changing their door, the player wins the car.
- The player chooses the door hiding goat №2. The host shows goat №1. By changing their door, the player wins the car.
- The player chooses the door hiding the car. The host shows one of the two goats. By switching, the player loses.

After understanding this simple formulation, we are ready to generalize the problem to having a total of N doors, of which just $p < N - 1$ doors are opened by the host. In 1975, D. L. Ferguson showed that switching now wins with probability $\frac{1}{N} \cdot \frac{N-1}{N-p-1}$. This probability is always greater than $\frac{1}{N}$, therefore the player should always prefer to switch the door, even if the host opens only a single door ($p = 1$). The greater is N and $N - p$, the lesser is the chance of winning, which will tend to zero. If, on the other hand, the host opens all of the doors but one ($p = N - 2$), the advantage of winning should always increase as N increases. The goal of this last exercise is to encoding the Generalized Monty Hall game play in a slight variation of the game from the previous use case scenario, which is the following:

Listing 10: probabilities/FourthAndFifthScenario.cs

```

1 using System;
2 using System.Linq;
3
4
5
```

```

6 namespace ConsoleApp2.probabilities {
7     class FourthAndFifthScenario {
8
9         int nDoors;
10        int nStates;
11        GeneralMontyHallProblem[] states;
12
13        public FourthAndFifthScenario(int nDoors, int nStates, int seed, double p = 0.8) {
14            this.nDoors = nDoors;
15            this.nStates = nStates;
16            int pInt = Math.Min((int)Math.Round(p * ((double)nDoors-2)), nDoors-2);
17            states = Enumerable.Range(0, nStates).Select(i => new GeneralMontyHallProblem(i+1, seed+i, pInt,
18                ↪ nDoors)).ToArray();
19        }
20
21        public ulong playGame(bool runMontyHallGame = true, bool debug = false) {
22            ulong countAttempts = 0;
23            int[] attempt = new int[nStates];
24            bool correctAttempt = true;
25
26            do {
27                correctAttempt = true;
28                for (int i = 0; i < nStates; i++) {
29                    attempt[i] = states[i].pickADoor(runMontyHallGame);
30                    correctAttempt = correctAttempt && states[i].isCorrectDoor(attempt[i]);
31                    states[i].changeCorrectDoor();// Saving up the iteration time!
32                }
33                if (debug) Console.WriteLine("[{0}]\n\n", string.Join(",_", attempt.Select(i => i.ToString())));
34                countAttempts++;
35            } while (!correctAttempt);
36
37            return countAttempts;
38        }
39    }

```

Each run of the game considers a given amount of doors (line 14) associated to each of the `nStates` rooms (line 15). We also provide the game with the total amount of doors to be showed by the host to the player, which is expressed as a ratio over the total number of the doors (line 16). We also define a `states` array, which associates to each room an instance to the Generalized Monty Hall problem, which will be an object of type `GeneralMontyHallProblem`. For each of the rooms i , the constructor of such an object should accept: *(i)* the room $i + 1$ the player progresses to after either winning or losing the game *(ii)* the seed associated to the host's random number generator, *(iii)* the number of doors `pInt` to be showed by the host, and *(iv)* the total number of the doors for each room.

Each instance of the `GeneralMontyHallProblem` should support at least three `public` actions: *(i)* `pickADoor`, which lets the automated agent pick a door and, if `runMontyHallGame` is set to true, it also runs the General Monty Hall Problem after which the player will always choose a different door, *(ii)* `isCorrectDoor` checking whether the door id passed as an argument is a correct door choice, and *(iii)* `changeCorrectDoor` for preparing a fresh new winning door for the next game iteration. By doing so, we can make the player iterate over all the possible rooms within the game, from the first to the last (line 27), and early detect whether the player did a wrong choice (line 29). The game will then terminate if and only if the player would be able to guess all the correct doors for each state of the game (line 34). We also count the number of attempts required by the current seed configuration before finding the correct door (`countAttempts`).

The solution for the automated game mechanics is given in the following Listing: we might favourably exploit the same `Door` class from the previous uses cases for each door, and an array of such doors to represent the whole set of doors in the current game instance (line 9). Each door should both be unlocked (line 37) and lead towards the next state (line 38). We also need to create a random number generator (line 26), that can be favourably exploited for generating a new (line 68) and different (line 66) winning (line 70) door configuration from the previous one (line 29), for letting the host pick the p doors to be showed (line 85-90), and for the user to choose one door among the remaining ones (line 98).

Listing 11: probabilities/GeneralMontyHallProblem.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.Linq;
5
6 namespace ConsoleApp2.proBABILITIES {
7     class GeneralMontyHallProblem {
8
9         Door[] stateDoors;
10        int correctDoor;
11        int nDoors;
12        int p;
13        Random generator;
14        Utils.UniformRandom ur;
15        Utils.UniformRandom uRem;
16
17        /// <summary>
18        /// Initializing a state, by setting up an instance of the general MontyHallProblem
19        /// </summary>
20        /// <param name="correctDoor">offset of the correct door</param>
21        /// <param name="nextReachableState">next Reachable State from the current state</param>
22        /// <param name="currentStateSeed">Seed associated to the random number generator for the current state
23        /// <param name="p">Number of doors to be showed while running the MontyHall problem</param>
24        /// <param name="nDoors">Number of doors associated to the state</param>
25        public GeneralMontyHallProblem(int nextReachableState, int currentStateSeed, int p = 4, int nDoors = 6,
26            bool debug = false) {
27            generator = new Random(currentStateSeed);
28            ur = new Utils.UniformRandom(0, nDoors - 1);
29            uRem = new Utils.UniformRandom(0, p - 1);
30            correctDoor = ur.nextInt(ref generator);
31            if (debug) Console.WriteLine("Setting_Correct_Door_={0}", correctDoor);
32
33            Debug.Assert(p < nDoors - 1, "The_number_of_the_showed_rooms_should_be_less_than_the_number_of_the_
34                doors_1!");
35            stateDoors = Enumerable.Range(0, nDoors).Select(i => new Door()).ToArray();
36            long j = 0;
37            Debug.Assert(correctDoor < nDoors);
38            foreach (Door refD in stateDoors) {
39                refD.locked = false;
40                refD.reachable_state = nextReachableState;
41                if (j == correctDoor) {
42                    refD.wrong_door = false;
43                }
44                j++;
45            }
46
47            this.nDoors = nDoors;
48            this.p = p;
49        }
50
51        /// <summary>
52        /// Checks whether i is the correct door
53        /// </summary>
54        /// <param name="i"></param>
55        /// <returns></returns>
56        public bool isCorrectDoor(int i) {
57            return i == correctDoor;
58        }
59
60        /// <summary>
61        /// Change the correct door associated to the current state
62        /// </summary>
63        public void changeCorrectDoor(bool debug = false)
64        {
65            int nextDoor = 0;
66            do {
67                nextDoor = ur.nextInt(ref generator);
68            } while (nextDoor == correctDoor);
69            stateDoors[correctDoor].wrong_door = true;
70            stateDoors[nextDoor].wrong_door = true;
71            if (debug) Console.WriteLine("Changing_Correct_Door_={0}", nextDoor);
72        }
73    }
74 }

```

```

70     correctDoor = nextDoor;
71 }
72
73 /// <summary>
74 /// Plays the game and picks a door
75 /// </summary>
76 /// <param name="runMontyPythonGame"></param>
77 /// <returns></returns>
78 public int pickADoor(bool runMontyHallGame = true) {
79
80     // The player picks a door
81     int pickDoor = ur.nextInt(ref generator);
82
83     if (runMontyHallGame) {
84         // Getting other p wrong doors, that should be different from the previous choice by the user
85         HashSet<int> choices = new HashSet<int>();
86         do {
87             var nextDoor = ur.nextInt(ref generator);
88             if ((nextDoor != pickDoor) && (nextDoor != correctDoor))
89                 choices.Add(nextDoor);
90         } while (choices.Count < p);
91
92         // Retrieveing the remaining N-p-1 doors, and choosing one of it
93         int nextChoiceOffset = uRem.nextInt(ref generator);
94         pickDoor = -1;
95         for (int i = 0; (i < nDoors) && (nextChoiceOffset >= 0); i++)
96         {
97             if ((!choices.Contains(i)) && (pickDoor != correctDoor)) {
98                 pickDoor = i;
99                 nextChoiceOffset--;
100             }
101         }
102         choices.Clear();
103         Debug.Assert(pickDoor >= 0);
104     }
105
106     return pickDoor;
107 }
108 }
109 }

```

Additional Exercises

1. If we denote a random variable X which is **uniformly distributed** within the interval $[a, b]$ as $\mathcal{U}(a, b)$, then we can write that $\mathcal{U}(a, b) = a + \mathcal{U}(0, 1) \cdot (b - a)$. By remembering the code for the `UniformRandom` class, take now a look at Example 2: how could we possibly code a class providing the sum of the outcome of two or more dices? Self-test your solution by checking whether the resulting implementation provides a distribution similar to the one in Figure 2.
2. Extend the formal notion of a DFA so to determine whether each edge represents a locked door, a wrong door, or if the same door was previously visited by the user
3. ★★ The knowledge given in this tutorial should be sufficient to understand the code in `goap/structures/WeightedMultiGraph.cs`. If you are not able to understand this code immediately, don't worry, we are going to discuss it in one of the following tutorials.
4. ★★★ Try to model the 2nd scenario without any state information and just try to stick to the formal definition of a Finite Automata. How many states are you going to need?
5. ★★★ Try to generalize the code for the 3rd scenario so to allow an arbitrary number of door and states. Furthermore, you could also try to generate a new set of correct doors at each new run of `testRandomCaseScenario`.