Phantom of the opera
AI

version 1.1

Philippe BOUTTEREUX

1 - The algorithm

The minimax algorithm is probably the most efficient algorithm for making decisions in this kind of game for several reasons.
The fact that cards are picked randomly at the beginning of each turn and the complexity of some of the characters' powers makes it very difficult (sometime impossible) to hardcode a strategy. Given this fact, the minimax algorithm is not only efficient, it is also and most importantly the easiest way of making an AI for this game. Furthermore, mixing it with some logic can make pretty satisfying results.

Even though, as said earlier, the minimax algorithm is very efficient, it also has its downfalls:

- Given the fact that cards are picked randomly every turn, one can only foresee efficiently up to one turn (2 sets, that is 8 moves)
- The time complexity is an important factor for a game with many play possibilities. In fact, here is the average number of node for a tree that would include a whole set (4 moves)

$$\prod_{n=1}^{4} (6.2 \cdot n) = 24.8 \cdot 18.6 \cdot 6.2 \cdot 12.4 = 35463.20$$

  In this expression, 6.2 is the average number of possibilities for a character (moves + powers - detailed in part 3). We start with 4 available characters and with each recursion, we remove one of them (because one character can't move twice in the same set). Generating a tree this big is obviously pretty long, which means it would be even crazier to go further than a set.
- The goal of the minimax algorithm is to predict the moves of the opponent in order to minimize the loss. However, as the inspector, you don't have access to the position of the ghost (you're supposed to find it). Given the fact that the ghost will certainly use this information to make his plays, it becomes much harder to guess them.

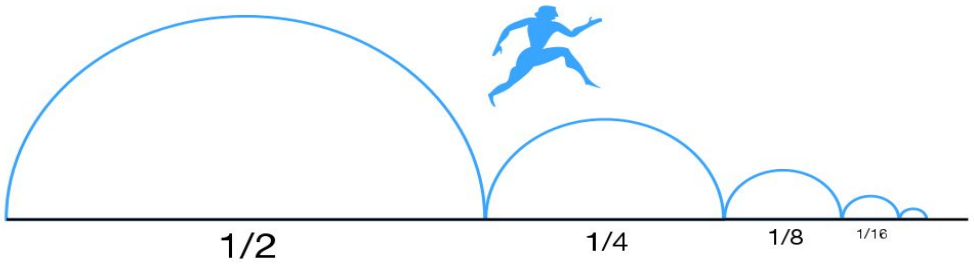
Lucky for us, most of the characters' powers can be trimmed down by logic (since we won't try to foresee further than a set, most power uses are pointless).

The inspector and the ghost AI use pretty much the same strategy (they share most of the code). The only differences being the goal and the informations they have access to.

In order to achieve these goals, we give each leaf node a weight (gain). Using these weights, we will be able to follow the most interesting path.
The computation of the weight depends if we are playing as inspector or ghost.

1.1 - Inspector's goal

The goal of the inspector it to apply dichotomy on all the suspect characters.



It means he want to arrange the characters so that at the end of every set, half of them are grouped and half are isolated (or in the blackout room).
His goal, by doing this, is to cut in half the number of suspects every set. If done perfectly (with luck on our side!), it will minimize the loss and the ghost will be found in 4 sets (or 2 rounds) at most. Again, the game being kind of random and fairly complicated, we don't expect it to be done perfectly. Still it seems to be the safest playstyle for the inspector.
The gain calculation is like so:

gain = 8 - abs((isolated) - (grouped))

The result will be between 0 (very bad) and 8 (very good). If most people are grouped, we add 0.1 to prioritize this state.

1.2 - ghost's goal

The ghost's goal is almost the opposite of the inspector's but he has access to one more information: where the ghost character is.
With that in mind, he will want to have as most people as possible either grouped or isolated. Choosing between grouping or isolating characters is of few importance because the gain if the ghost is isolated is only 1. What is far more important is having the ghost in the most numerous side. Say, we have 6 people isolated and only two grouped, having the ghost in the isolated side would mean almost certain loss. This is the thing to keep in mind while playing ghost. The gain computation is :

gain = (ghost side) - other side)

Where the sides are "grouped" and "isolated".
If the ghost side is "isolated", we add .1 to prioritize this state.
The result will be between -6 (automatic loss) and 8 (perfect).

1.3 - Achieving these goals

It is important, when making the strategy, to keep in mind the very purpose of the minimax algorithm: foreseeing the enemy's move and minimizing loss. It means that for a set, depending on whether we are the inspector or the ghost and in which part of a round we are, we will need to use the inspector's gain calculator or the ghost's. For instance, if we play as the inspector and it is the first set, we will need to maximize gain by using the calculators in this order:

Inspector, Ghost, Ghost, Inspector

Also, as we said earlier, the inspector doesn't have access to the identity of the ghost, so he will use a custom gain calculator to try and mimic the ghost's playstyle. It works like so:

gain = abs((isolated) - (grouped))

Again, the result will vary from 0 to 8 (8 being optimal for the ghost) and we will add 0.1 if most people are isolated.

2 - Decision tree generation

In the code of the AI, The minimax algorithm will be represented as a decision tree. It will be composed of Nodes of different types with different purposes.
Here are the basic nodes of our tree:

2.1 - Node

This is the abstract (base) class for every node. It contains the attributes and functions that will allow us to navigate through our tree, store informations in it and retrieve them.
It is not supposed to be used as is.

2.2 - RootNode

The Root Node is the base of our tree. It will loop through all available options and generate the branches. It doesn't compute or store any information, it is only functional.

2.3 - CharacterNode

This is the base class for characters. It prepares some of the data needed by characters to use their moves and powers. Just as Node, this class is not to be used alone. It serves as base for the characters implementations.

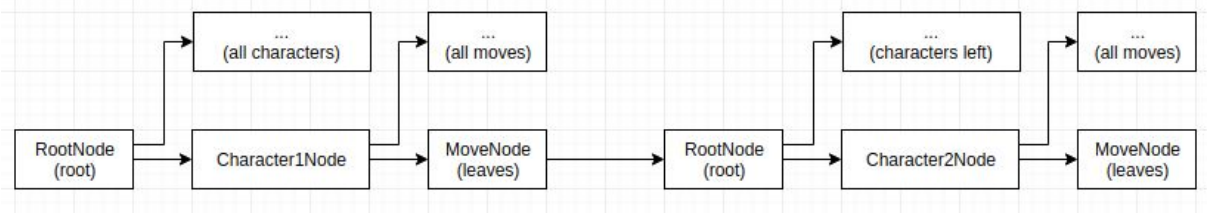
2.4 - DefaultChNode

While all the characters' powers aren't implemented in the game, this class is here to handle the ones that are not done yet. It never uses powers and simply move to an adjacent room (note that the choice of the move has nothing to do with randomness. It may be a default but it still computes all the possible ways to find the best one, only avoiding power).

2.4 - MoveNode

The Move nodes can be thought as the leaves of the tree but there is a trick: they will themselves call a RootNode which will in turn, call a CharacterNode and so on until the end of the set. The characters' powers will look like MoveNodes and be at the same level. Their interactions will depend on the character and be discussed in part 3 of this document.

Here is a short drawing of what the tree looks like (obviously, it would take a lot of space to show the tree for an entire set).



3 - Powers implementations

In order to build the decision tree, it was necessary to implement the behaviour of every characters' powers. Here are the ways they have been implemented. We will also see the complexity (the number of possible moves for a character) and, if needed, a schema of the tree branches. Some of the powers aren't implemented yet. Please refer to the README for an updated list of implemented powers.

constants for the complexity computation:

- C: 8 (the number of characters)
- R: 10 (the number of rooms)
- P: 2.2 (the average number of rooms any character can go)

3.1 Raoul De Chagny

Raoul's power doesn't need any extra node. It allows us to pick a card and (maybe) get a free alibi. This possibility is taken into account by adding 0.5 to the final score of the node. It means that it will have priority over another path with the same final score but not over a path with an assured higher score. Note that the only interest for the ghost is to deny alibies to the inspector. However, the behaviour of the node stays the same.

Complexity: P

3.2 Meg Giry

Meg's power is the easiest to implement. she simply has access to move paths than the others. No additional nodes needed.

Complexity: 3.3

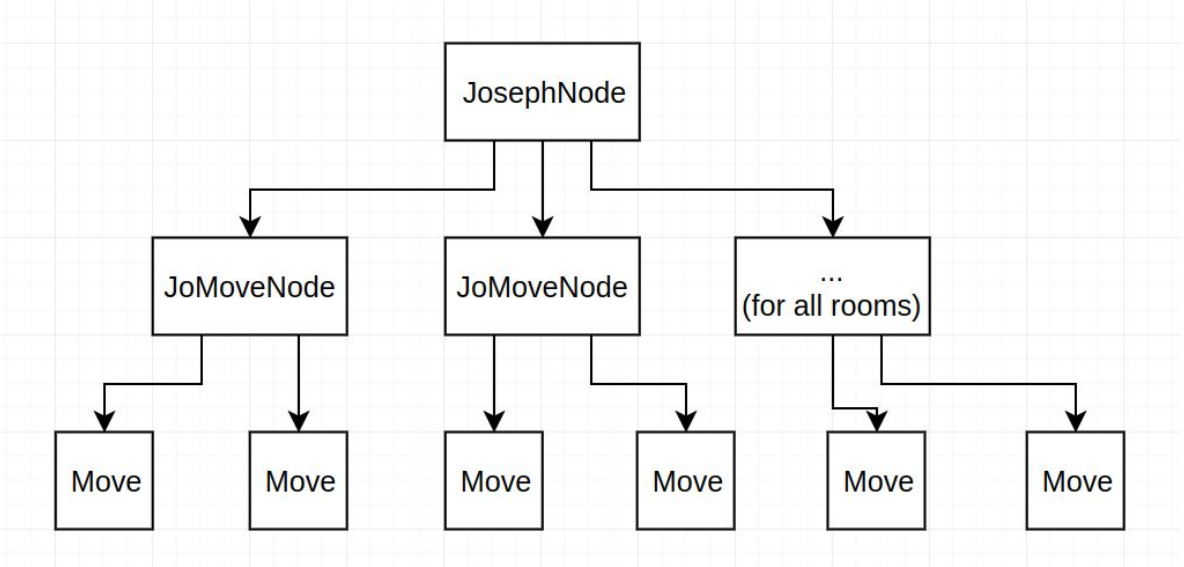
3.3 Ms Giry

For Ms Giry, there are two possible ways of handling the position of the lock. Either trying to put it on every room (it makes the decision tree much bigger) or looking for it in a logical way (depending on the people in each room and their status (it is less powerful but also less consuming).
I did the latter.

Complexity: P

3.4 Joseph Buquet

Just as Ms Giry, the power of Joseph can either be included in the tree or hardcoded. However, this power is very important, so I decided to use the first option. It results in high complexity but also high efficiency.



Complexity: RxP

3.5 Christine Daaé

For christine, we will also create nodes for every possible power uses because the added complexity is not too big.

Complexity: 2xP (every possible path with AND without using the power)

3.6 Mr. Moncharmin

This power can, in some cases, be pretty expensive to implement as nodes of the tree because of the possibility to treat each people differently. Furthermore, I think his power is not very… powerful. For those reasons, we will simply insert in the tree the node saying he uses his power and then resolve the power's action in a logical way, to isolate or regroup people by taking care of who else will play before the end of the set.

Complexity: P * 2 (every possible path with AND without using the power).

3.7 Mr. Richard

Mr richard can permute himself with any of the other seven characters in the game. However, swapping with a character that wont move anyway doesn't give any short-term gain, so we can trim down to "any character that has yet to move during the set (max. 3).

Complexity: P + (number of characters that can still move during the set)

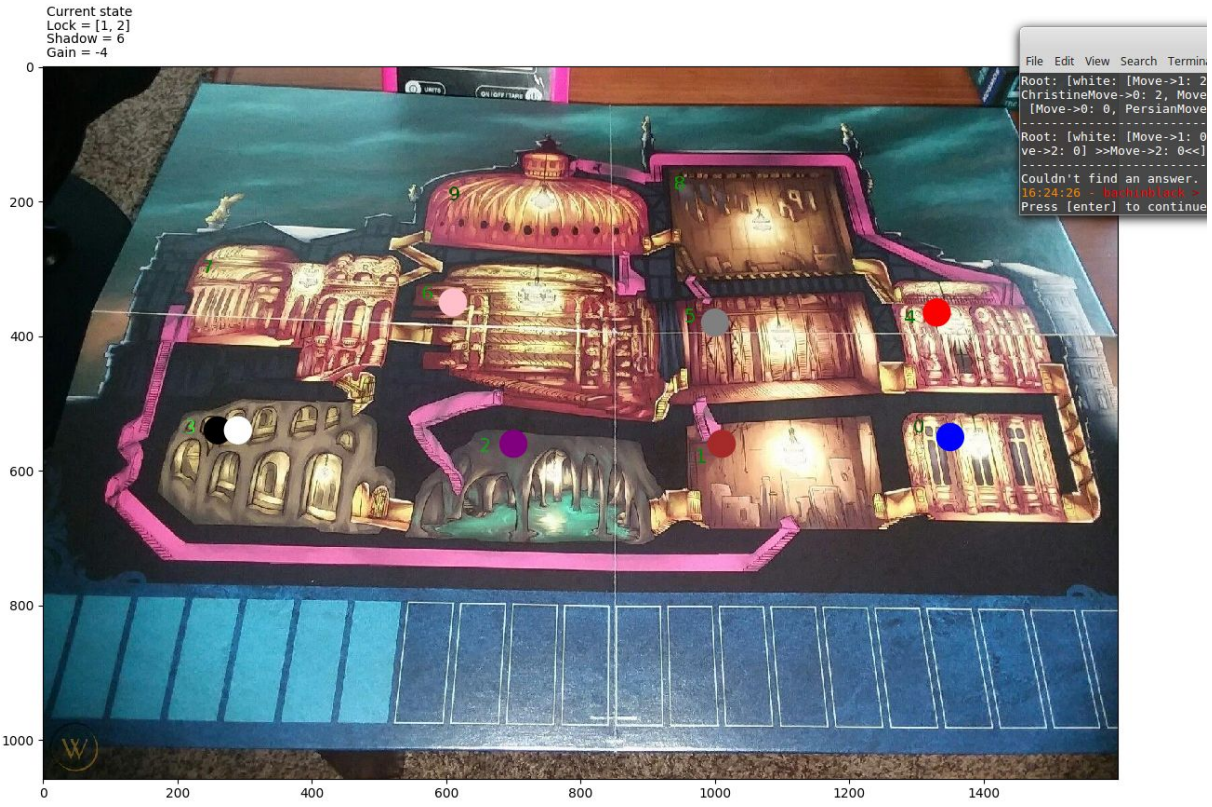
3.8 The Persian

The power of the Persian is one of those that don't add up to too much complexity. Therefore, it also creates nodes for every possible cases. He will try to move to every locations with every possible characters. However, it could be meaningless to do this. The power can and will certainly be trimmed down to "every people that can still move during the set".

Complexity: P * (number of characters in the same room)

4 - Graphical visualization of the game

In order to debug the game, the most efficient way was to make it graphical :



It was made with matplotlib and allows us to see the position of every character in the board (colored circles), along with the lock and blackout's position, the current gain and the last move made. Characters that are not suspect anymore are represented by squares instead of circles.
When it opens, set focus on the terminal and press [enter] to see all the possibilities, one by one.

To activate or deactivate it, simply set the call argument to display.init_debug to True or False (player.py, l.19).