

Clue in Prolog

– A Didactic Example –

Luciano Santos

September, 2016

1 Introduction

This is the documentation for a simple script in SWI-Prolog that plays the game Clue¹. This implementation follows a didactic approach, not aimed at creating an advanced AI system that employs complex strategies and human behaviour models to master the game. It simply illustrates how a declarative language can be used to play a relatively simple game based on a certain set of rules.

The implementation is based on the rules for the 2002 version of the game (see PDF on the root folder) and the board on Figure 1.



Figure 1: The game board.

The following principles were observed in this implementation to make it simple:

- no long-term planning – for each action and information received, the agent updates its 'knowledge base' and, on each turn, it makes an independent decision based on the current knowledge, instead of following a planned route;
- no lucky guesses – the agent only makes an accusation if it's certain that it's true;
- no poker face – the agent only acts to acquire more information, and will not make a move or guess for the sole purpose of misleading other players;

¹<http://www.hasbro.com/en-us/toys-games/hasbro-games:clue> (accessed on August, 2016)

- no mind reading – the agent will not infer information from other players actions, except facts that can be logically proven; it will not try to predict how people would or should behave, however, it will assume that everyone will play strategically, e.g., if the player to the left has already shown a certain card before, that card will not be used on a next guess, because a smart player would keep showing the same card over and over again, even if she had a different one to show.

The sections below describe the rationale and the details of this implementation. Section 2 shows how the game board and the current position of each player is stored internally and how the agent finds the shortest path to a given goal. Section 3 describes how the knowledge acquired as the match progresses is represented, and how the game decides the action to take in each turn. Finally, Section 4 explains the predicates that allow the final user to initialize and subsequently interact with the agent.

2 Moving on the Board

The game board is seen internally as a grid of size 24×25 . As illustrated by Figure 2, coordinates are relative to the lower left corner, and start on 0.

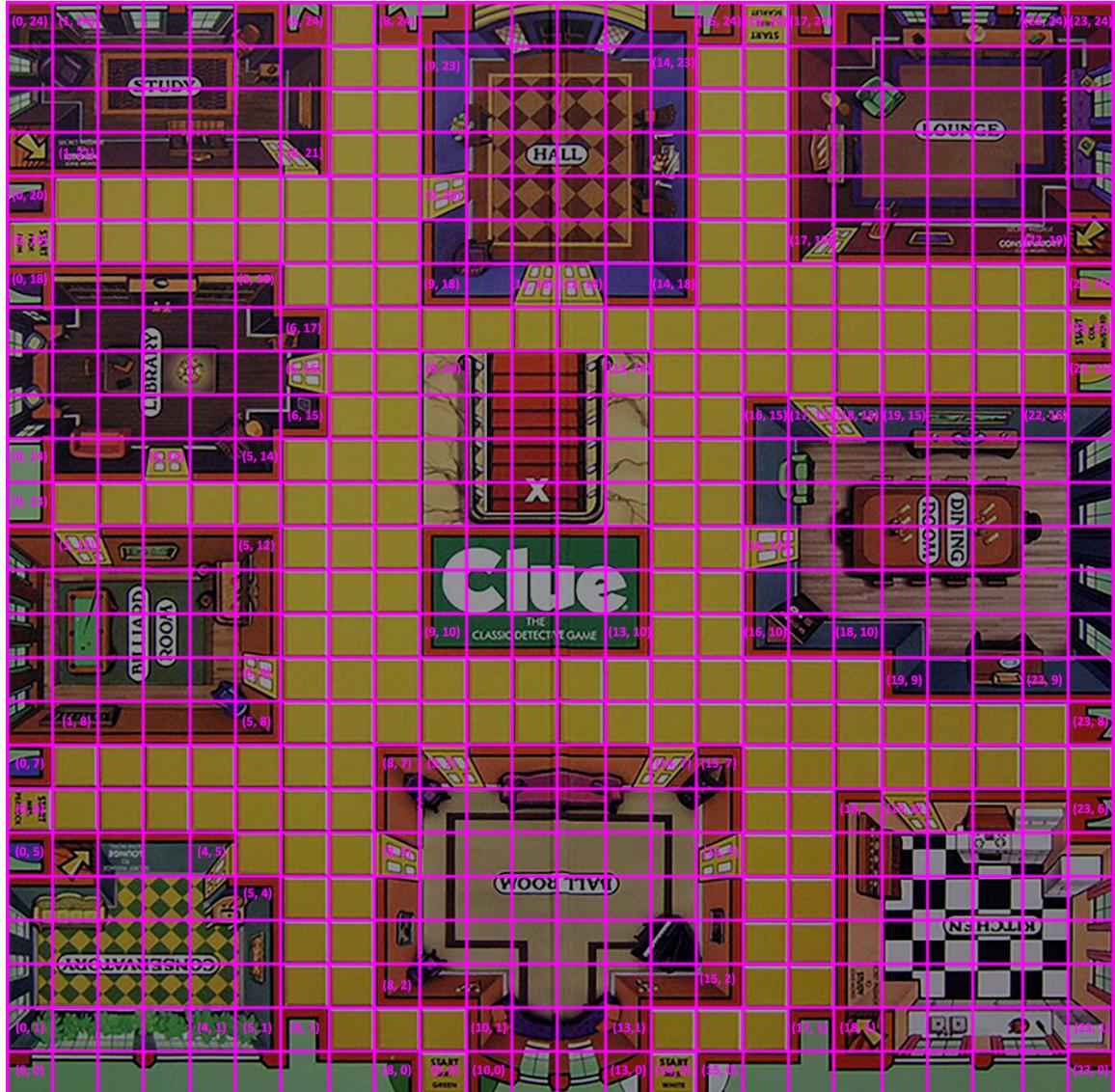


Figure 2: The game board.

2.1 Constraints

The following predicates describe the constraints that affect movements inside the board:

- `blocked/1` – is true for the tuple (X, Y) if it represents a location that is blocked, i.e., it's inside a room or in a wall;
- `door/2` – is true for the tuple (X, Y) and room R if it's possible to enter R from (X, Y) ;
- `passage/2` – is true for arguments $\text{Source}, \text{Dest}$ if there is a secret passage from room Source to room Dest ;
- `position/2 (dynamic)` – is true for the tuple (X, Y) and character C if (X, Y) is the current position of C on the board;

Based on the game rules, once inside the room, the position of the individual cell occupied by the player is irrelevant, it's as if each room is a "supercell". For that reason, it's only necessary to know if the player is inside a room and, if not, in which (free) cell she is. Also, because each room is accessible only from specific cells, and each of these cells gives access to one, and only one room, it's unnecessary to know if a given cell belongs to a specific room or wall, it's only important to know which are the access cells (i.e., doors) to which room, and if a given cell is free or not.

Since the blocked areas are more "regular", i.e., they are more easily described in terms of rectangles, it was a design choice to use a predicate `blocked` that defines if a certain position is blocked or not. The same end could be achieved with an opposite predicate `free`. Figure 7 shows the (static) declaration of the blocked cells in the board.

The predicate `door` says if a cell is an access cell, and to which room it gives access to. Since entering the room doesn't count as a step, it's enough to reach these access cells when finding a path and then switching the state to "inside the room". Figure 8 shows the (static) declaration of all the doors in the game.

The predicate `passage` is a special case, to represent the information that there are secret passages between the rooms in the (opposite) corners of the board. As shown in Figure 9, it is implemented as a commutative operation by using an auxiliary predicate.

Finally, the predicates `position` and `last_position` (this one will be used later) inform the current and the last turn's position of a player, respectively. For any atom n that is a valid player name, there will be exactly one fact that says "the position of n is (X, Y) ". The agent makes sure this holds true by defining the start position shown on the board once (Figure 10), and *retracting* and *asserting* that position any time a player's position changes, as will be explained on Section 3. When a player enters a room, her position becomes that room's name.

Other constraints, such as the board boundaries, are checked on the path finding algorithm, as described on the next section.

2.2 Path Finding

One of the key actions the agent must perform is movement. The first step necessary to move on the board is to answer the question: "where can I go from my current position?". If the player is in a cell adjacent to a door, she can enter the room; if she is in a corner room, she could use a secret passage. However, to get to the point where she could enter a room, the player must first reach a door, and using a secret passage is a trivial action. Thus, right now, only the problem of moving from a free cell to another free cell will be addressed. In order to solve that problem, the agent must determine the shortest path between two points.

As the game rules state, if the player is currently in any given cell, she can only move to another empty cell vertically or horizontally. To express that relationship, the predicate `neighbor/2` is defined (Figure 3). It will be true for points (X_s, Y_s) , (X_n, Y_n) if (X_n, Y_n) is exactly one cell away from (X_s, Y_s) , moving either vertically or horizontally (but not both).

```
%% neighbor((Xs, Ys), (Xn, Yn)) - <Xn, Yn> is neighbor of <Xs, Ys>.
neighbor((Xs, Ys), (Right, Ys)) :- Right is Xs + 1.
neighbor((Xs, Ys), (Left, Ys)) :- Left is Xs - 1.
neighbor((Xs, Ys), (Xs, Up)) :- Up is Ys + 1.
neighbor((Xs, Ys), (Xs, Down)) :- Down is Ys - 1.
```

Figure 3: The declaration of the predicate `neighbor`.

Now, besides checking if a certain cell is a neighbor, it must also be possible to actually move there, *i.e.*, it must be inside the boundaries of the board and not occupied by any other player or blocked, as previously defined. All these cases are summarized by the predicate `is_free/1`, shown in Figure 4.

```
%% is_free((X, Y)) - the position <X, Y> can bee occupied by a character.
is_free((X, Y)) :-
    X >= 0, X <= 23, Y >= 0, Y <= 24, % inside the board
    \+ position((X, Y), _), % not currently occupied by anyone
    \+ blocked((X, Y)). % not a room or a wall
```

Figure 4: The declaration of the predicate `is_free`.

Once all the constraints on movement between two adjacent cells have been represented, it's possible to implement the algorithm to find the shortest path between two points on the map, if such a path exists. More precisely, the algorithm here implemented finds the shortest path between a given free cell (X_s, Y_s) and the closest door. Subsequent backtracking on the predicate will return all the doors of the board, ascendingly ordered by path length.

The simplest way to tackle this task is using BFS (Breadth-first Search). The idea is simple: inspect the source node; then all the nodes exactly 1 step away from the source node; then all the nodes exactly 2 steps away from the source node; and so on. This ensures that, if a path exists, it will be found and will be a shortest path. If the backtracking mechanism continues from the state where it found a path, the next door found will have the next shortest path.

To make sure that the nodes are inspected in the correct order, a queue is used. This queue begins with the source node. At each iteration, the node at the head of the queue is removed and inspected. If it's not a door, then its adjacent nodes are enqueued and the search continues. To prevent the search from recursing infinitely, it's necessary to keep a set of all nodes seen so far, so only adjacent nodes not yet inspected are enqueued.

Notice that this algorithm and the way the constraints are defined respect the game rules that state “*you must not enter or land on a square that's already occupied*”, “*you may not [...] enter the same square twice on the same turn*”, “*you may not pass through a door that's blocked by an opponent's character*” and “*when you pass through a door, do not count the doorway itself as space*”. This derives from the facts that every player's positions are considered blocked cells, that BFS never moves to a previously seen cell and that doors are represented by the only cell that gives access to them, and it's necessary and sufficient to reach them to enter a room.

The code for this search algorithm is shown in Figure 5. The predicate `closest_door/3` receives a source point and unifies the name of the room with the closest door and the path

to that door as a list of points. This predicate itself only sets up the initial values for the algorithm – the queue and the “seen nodes” set are both initialized to a unitary list containing the source point – and calls the auxiliary predicate `closes_door_aux/4` to actually perform the search.

The base case for this predicate is when the head of the queue is a door. If that happens, it will unify to the found door’s room and the path is a unitary list containing just the door. The recursive step will extract the head of the queue, enqueue all adjacent cells that are free and have not yet been seen (described by `valid_adjacent/3`), mark all the neighbor cells are seen, and recursively continue walking on the list. As the recursive calls return, the path is built by inserting the inspected node on the head of the result list.

```
%% closest_door([Xs, Ys], Room, Path) - starting at <Xs, Ys>, finds
%% the closest Room door and the Path to it
unseen_neighbor((X, Y), (Xd, Yd), Seen) :-  
    neighbor((X, Y), (Xd, Yd)),  
    \+ member((Xd, Yd), Seen).  
valid_adjacent((X, Y), (Xd, Yd), Seen) :-  
    unseen_neighbor((X, Y), (Xd, Yd), Seen),  
    is_free((Xd, Yd)).  
closest_door_aux(Room, [Head], [Head|_], _) :- door(Head, Room).  
closest_door_aux(Room, [(X, Y)|PTail], [(X, Y)|QTail], Seen) :-  
    % enqueues all non-seen adjacents to which it's possible to move  
    findall((Xd, Yd), valid_adjacent((X, Y), (Xd, Yd), Seen), ValidAdjacents),  
    append(QTail, ValidAdjacents, NewQueue),  
    % stores all adjacent nodes as seen, to cut the search recursion  
    findall((Xa, Ya), unseen_neighbor((X, Y), (Xa, Ya), Seen), Neighbors),  
    append(Neighbors, Seen, NewSeen),  
    closest_door_aux(Room, PTail, NewQueue, NewSeen). % recursive definition  
closest_door([Xs, Ys], Room, Path) :-  
    closest_door_aux(Room, Path, [(Xs, Ys)], [(Xs, Ys)]).
```

Figure 5: The breadth-first search used to find the closest door, given a source cell.

Even though this algorithm finds the path between two cells on the board, the agent actually needs to solve the more general problem of finding the closest doors when departing from multiple source cells. That’s because, if the agent is inside a room, there could be multiple exits to choose from, thus, it should find the best option taking into account all these exits.

The naïve approach would be to find lists with all the paths for each exit, merge these lists while ordering the elements (paths) by length and, finally, traverse the result. However, that solution is both inefficient and *imperative*, in the sense that it tells the steps to solve the problem, instead of taking advantage of the declarative paradigm.

What happens if the queue in the previous solution is initialized with multiple source cells, instead of just one?

- when the first element of the queue is inspected, all its adjacent nodes will be enqueue **after** all the other sources, which are currently on the queue (as long as those adjacent nodes aren’t sources themselves);
- when the next element is inspected (another source), once again, all its *not already on the queue* adjacents will also be enqueue;
- after all the sources nodes are inspected first, the next element to be inspected is exactly one step away from one of the sources; actually, all the elements that are exactly one step away from at least one of the sources will be inspected next, and their adjacent nodes will be enqueue;

- by analogy, all the elements that are exactly two steps away from **at least one** of the sources will be analyzed next, and so on...

Ergo, it's possible to conclude that, just by initializing the queue with the multiple sources, the order in which the elements are inspected still ensures that doors are searched *breadth-first* and, as a consequence, the doors are found in order, from the closest to the farthest, only now the distance applies to any one of the possible sources.

There's one problem though: in the previous algorithm, since there was only one source, the path could be rebuilt implicitly within the backtracking mechanism. The base of the recursion was the door, and the recursive steps would attach one cell to the head of the resulting list, until the caller of the predicate would have the whole path ready. With multiple sources, however, it's necessary to explicitly store, for each cell that is reached, that cell's parent, *i.e.*, the cell *from which* it was reached.

```
%% closest_door([Xs, Ys], Room, Path) - starting at <Xs, Ys>, finds
%% the closest Room door and the Path to it
unseen_neighbor((X, Y), (Xd, Yd), Seen) :-
    neighbor((X, Y), (Xd, Yd)),
    \+ member(((Xd, Yd), _), Seen).
valid_adjacent((X, Y), (Xd, Yd), Seen) :-
    unseen_neighbor((X, Y), (Xd, Yd), Seen),
    is_free((Xd, Yd)).
add_parent([], _, []).
add_parent([Node|NodesTail], Parent, [(Node, Parent)|LinkedTail]) :-
    add_parent(NodesTail, Parent, LinkedTail).
build_path(Head, Seen, [Head]) :- member((Head, nil), Seen).
build_path(Head, Seen, [Head|Tail]) :-
    member((Head, Parent), Seen),
    build_path(Parent, Seen, Tail).
closest_door_aux(Room, Path, [Head|_], Seen) :-
    door(Head, Room),
    build_path(Head, Seen, InversePath),
    reverse(InversePath, Path),
    print(Path), nl.
closest_door_aux(Room, Path, [(X, Y)|QTail], Seen) :-
    % enqueues all non-seen adjacents to which it's possible to move
    findall((Xd, Yd), valid_adjacent((X, Y), (Xd, Yd), Seen), ValidAdjacents),
    append(QTail, ValidAdjacents, NewQueue),
    % stores all adjacent nodes as seen, to cut the search recursion
    findall((Xa, Ya), unseen_neighbor((X, Y), (Xa, Ya), Seen), Neighbors),
    add_parent(Neighbors, (X, Y), LinkedNeighbors),
    append(LinkedNeighbors, Seen, NewSeen),
    closest_door_aux(Room, Path, NewQueue, NewSeen). % recursive definition
closest_door([Xs, Ys], Room, Path) :-
    closest_door_aux(Room, Path, [(Xs, Ys)], [(Xs, Ys), nil]).
closest_door(SourceRoom, TargetRoom, Path) :-
    valid_rooms(ValidRooms), member(SourceRoom, ValidRooms),
    findall(Exit, door(Exit, SourceRoom), Exits),
    add_parent(Exits, nil, LinkedExits),
    closest_door_aux(TargetRoom, Path, Exits, LinkedExits).
```

Figure 6: The breadth-first search used to find the closest door, given multiple source cells.

Figure 6 shows the new version of the algorithm that includes these changes. Now, the Seen set elements are tuples (Cell, Parent) that store not only the coordinate of the cells that were already seen, but also their parent (which can be the coordinates of a previously seen cell or `nil` for the source cells). All predicates were updated to handle this new representation.

The auxiliary predicate `add_parent/3` simply receives a list Nodes and an element Parent and generates a new list with each element N of Nodes in a tuple (N, Parent).

Finally, the new recursive predicate `build_path/3` is used to explicitly rebuild the path when a door is reached, given the set of seen nodes (and their parents). The predicate inspects the Head element using the Seen set and generates the Path as a list; the base case happens when the parent of Head is `nil`, which generates the unitary list; the recursive case adds Head

to the list and inspects its non-nil parent. Since this predicate is called when a door is found, the path is actually generated in the reverse order, so the resulting list is reversed before being returned to the original caller.

```
%% blocked((X, Y)) - point <X, Y> is inside a room or is a wall
% conservatory
blocked((X, Y)) :- Y =:= 0, X >= 0, X <= 8.
blocked((X, Y)) :- X >= 0, X <= 4, Y >= 1, Y <= 5.
blocked((X, Y)) :- X =:= 5, Y >= 1, Y <= 4.
blocked((6, 1)).
% ball room
blocked((X, Y)) :- X >= 10, X <= 13, Y >= 0, Y <= 1.
blocked((X, Y)) :- X >= 8, X <= 15, Y >= 2, Y <= 7.
% kitchen
blocked((X, Y)) :- Y =:= 0, X >= 15, X <= 23.
blocked((17, 1)).
blocked((X, Y)) :- X >= 18, X <= 23, Y >= 1, Y <= 6.
% dining room
blocked((X, Y)) :- X >= 16, X <= 18, Y >= 10, Y <= 15.
blocked((X, Y)) :- X >= 19, X <= 22, Y >= 9, Y <= 15.
blocked((X, Y)) :- X =:= 23, Y >= 8, Y <= 16.
% lounge
blocked((X, Y)) :- X >= 17, X <= 22, Y >= 19, Y <= 24.
blocked((X, Y)) :- X =:= 23, Y >= 18, Y <= 24.
% hall
blocked((X, Y)) :- X >= 9, X <= 14, Y >= 18, Y <= 23.
blocked((X, Y)) :- Y =:= 24, X >= 8, X <= 15.
% study
blocked((X, Y)) :- X =:= 0, Y >= 20, Y <= 24.
blocked((X, Y)) :- X >= 1, X <= 6, Y >= 21, Y <= 24.
% library
blocked((X, Y)) :- X >= 0, X <= 5, Y >= 14, Y <= 18.
blocked((X, Y)) :- X =:= 6, Y >= 15, Y <= 17.
% billiard room
blocked((X, Y)) :- X =:= 0, Y >= 7, Y <= 13.
blocked((X, Y)) :- X >= 1, X <= 5, Y >= 8, Y <= 12.
% stairs
blocked((X, Y)) :- X >= 9, X <= 13, Y >= 10, Y <= 16.
```

Figure 7: The declaration of the predicate `blocked`.

```
%% door((X, Y), R) - there's a door to room R from point <X, Y>
door((5, 5), 'conservatory').
door((7, 5), 'ball room').
door((16, 5), 'ball room').
door((9, 8), 'ball room').
door((14, 8), 'ball room').
door((19, 7), 'kitchen').
door((15, 12), 'dining room').
door((17, 16), 'dining room').
door((17, 18), 'lounge').
door((8, 20), 'hall').
door((11, 17), 'hall').
door((12, 17), 'hall').
door((6, 20), 'study').
door((3, 13), 'library').
door((7, 16), 'library').
door((1, 13), 'billiard room').
door((6, 9), 'billiard room').
```

Figure 8: The declaration of the predicate `door`.

```

%% passage(Src, Dst) - there's a passage from Src to Dst
passage_aux('conservatory', 'lounge').
passage_aux('kitchen', 'study').
passage(Src, Dst) :- passage_aux(Src, Dst) ; passage_aux(Dst, Src).

```

Figure 9: The declaration of the predicate `passage`.

```

%% position((X, Y), C) - the current position of character C is <X, Y>
%% here, it's initialized to the start position at the board.
%% this predicate will be "rewritten" whenever the script's own
%% character moves or it receives information that another character
%% moved.
:- dynamic position/2.
position((16, 24), 'scarlet').
position((23, 17), 'mustard').
position((14, 0), 'white').
position((9, 0), 'green').
position((0, 6), 'peacock').
position((0, 19), 'plum').

%% last_position((X, Y), C) - the position of character C was <X, Y> last turn.
%% it'll be updated as the script's own character moves or it receives
%% information that another character moved.
:- dynamic last_position/2.
last_position(nil, 'scarlet').
last_position(nil, 'mustard').
last_position(nil, 'white').
last_position(nil, 'green').
last_position(nil, 'peacock').
last_position(nil, 'plum').

```

Figure 10: The declaration of the predicate `position` and the initial position of all players.

3 Making Decisions

This section describes how the agent represents knowledge about the game internally (Section 3.1), and how it then uses that knowledge to make decisions in each turn (Section 3.2).

3.1 Representing the Game Data

The first data that need to be represented internally are the lists of valid rooms, characters and weapons. These (static) lists are initialized with the predicates in Figure 11 and used throughout the code. For simplicity, no (physical) player name is stored, instead, all players are referenced by their character's name. The list of valid characters is declared in the order that those characters would play, according to the rules. The agent's own character is represented by the dynamic predicate `my_char/1`, that is set once when a match starts.

```

%% valid_rooms(Rooms) - Rooms is the list of valid room names.
valid_rooms(
    [
        'conservatory',
        'ball room',
        'kitchen',
        'dining room',
        'lounge',
        'hall',
        'study',
        'library',
        'billiard room'
    ]
).

%% valid_chars(Chars) - Chars is the list of valid characters,
%% in the order they must play.
valid_chars(
    [
        'scarlet',
        'mustard',
        'white',
        'green',
        'peacock',
        'plum'
    ]
).

%% my_char(Char) - Char is the character of this agent.
:- dynamic my_char/1.

%% valid_weapons(Weapons) - Weapons is the list of valid weapons.
valid_weapons(
    [
        'rope',
        'pipe',
        'knife',
        'wrench',
        'candlestick',
        'pistol'
    ]
).

```

Figure 11: The lists of rooms, characters and weapons in the game; and the current player’s char.

Next, an important information required by the agent to make a move is “which cards have I seen?”. To win the game, a player uses a very simple elimination process: the total number of cards is known for each category – room, character and weapon; there’s exactly one card of each in the “confidential” envelope; if, for any of those, only a single card has not yet been seen, then it’s logical that it must be on the envelope.

Figure 12 shows the very straightforward predicates that represent the information of which cards have been shown, and by which player. This last information is necessary for the decision step described later. Notice that the player who has shown the card might be the agent itself, meaning that the card is on its hands.

The predicate `can_accuse/3` is true if the agent currently knows all the necessary information to make an accusation. It’s trivially defined as the case when the unknown rooms, characters and weapons lists are all unitary.

```

%% shown_char(Player, Char) - the player Player (could be myself)
%% has shown me the card for Char.
:- dynamic shown_char/2.

%% shown_room(Player, Room) - the player Player (could be myself)
%% has shown me the card for Room.
:- dynamic shown_room/2.

%% shown_weapon(Player, Weapon) - the player Player (could be myself)
%% has shown me the card for Weapon.
:- dynamic shown_weapon/2.

%% the next predicates generate the known and unknown lists
%% of characters, rooms and weapons, i.e., those whose cards
%% have and have not yet been shown so far
known_chars(KnownChars) :- findall(C, shown_char(_, C), KnownChars).
unknown_chars(UnknownChars) :-
    valid_chars(ValidChars),
    known_chars(KnownChars),
    subtract(ValidChars, KnownChars, UnknownChars).

known_rooms(KnownRooms) :- findall(R, shown_room(_, R), KnownRooms).
unknown_rooms(UnknownRooms) :-
    valid_rooms(ValidRooms),
    known_rooms(KnownRooms),
    subtract(ValidRooms, KnownRooms, UnknownRooms).

known_weapons(KnownWeapons) :- findall(W, shown_weapon(_, W), KnownWeapons).
unknown_weapons(UnknownWeapons) :-
    valid_weapons(ValidWeapons),
    known_weapons(KnownWeapons),
    subtract(ValidWeapons, KnownWeapons, UnknownWeapons).

can_accuse(Person, Room, Weapon) :-
    unknown_chars([Person]),
    unknown_rooms([Room]),
    unknown_weapons([Weapon]).

```

Figure 12: Representing which cards were shown, by whom.

3.2 Picking Actions

The basic three actions a player can take in the game are: moving, making suggestions and making accusations. Based on the game rules, the following definitions and constraints will be used to pick actions:

Definition 3.1 (Move) *A move is defined as changing the current position either by rolling the dice or by using a secret passage – remember that a player’s position can be the name of a room.*

Definition 3.2 (Entered a room) *A player is said to have entered a given room this turn if, at the end of her last turn, she was in a free cell or a different room and is now inside said room – including the case when she was moved there by another player.*

Definition 3.3 (Character and player rank) *A character’s rank is its position relative to the agent’s own character, following the sequence in which the characters must play their turns. The agent’s character has rank 0. The next character to the left of the agent’s character has rank 1, and so on, until all characters have received a rank.*

A player’s rank is the rank of the character it controls.

Constraint 3.1 *In a single turn, a player can make a move, a suggestion and an accusation, in that relative order, each one at most once per turn.*

Constraint 3.2 *A player can only make a suggestion in a given room if she entered that room this turn.*

Constraint 3.3 *A player can make an accusation from any position, as long as it's her turn.*

These statements describe in a simple way the set of constraints found in different sections of the game rules. In particular, notice that they cover the more complex rules that say that a player cannot re-enter the same room in a single turn and that when a character is referred in a suggestion or accusation, it's moved to the respective room and, in that case, its owner may make a suggestion in that room on her next turn, however, if the referred character was already in the room cited at the suggestion, then no transfer occurred and the "entered the room" rule doesn't apply.

Constraint 3.3, however, was explicitly stated here to choose an interpretation to an ambiguity on the rules:

- If the instructions are read until the point right before the section "Making an Accusation", the sentences "*If no one is able to prove your Suggestion false, you may either end your turn or make an Accusation now.*" and "*You may, if you wish, make a Suggestion followed by an Accusation on the same turn.*" would lead one to conclude that an accusation can only be made **if** and **after** a suggestion is made, and more specifically, if that suggestion is not proven false by anyone, with the corollary that the player would have to be inside a room to make an accusation.
- The section "Making an Accusation", however, by directly contrasting the act of making a Suggestion and making an Accusation, leads to the interpretation that an Accusation doesn't depend on a suggestion and could be done from any position at any time, as long as it's on the player's turn.

Analyzing the premise of the game and with the intention to make things simpler, the later interpretation is adopted.

By its own nature, the action picking algorithm is quite procedural. First, a very high level description will be given – it's summarized in Figure 13 – next, the predicates are explained in details.

The first thing the agent should do, since it can do it from any location, is to verify if there is currently enough information to make an accusation and, if so, pick that action. On all other cases, the agent uses a series of criteria to weight its options and decide how to proceed.

As either moving or making a suggestion requires picking a room, this is the next thing to be decided. Assume that, following Definition 3.2, the player just entered a room this turn. If that happens, the current room must have the highest weight in the decision, however, it can't be used immediately, as there's one case when it's not actually a good choice: when the next player has already shown this room on a previous turn. If that player is playing strategically (and it's assumed she is), using this room in a suggestion would bring no new information. If the player is currently inside a room but has not entered that room this turn, then it can't be used and must not be considered by the algorithm.

Based on this reasoning, the algorithm first sorts all rooms by path length from current position (if the player is currently inside a room, it will have the shortest distance and be the first on the list). Next, it traverses this list picking the first room that was not previously shown by the next player. It's highly probable that such a room exists, but, if not, the closest room is picked.

Let R be the picked room. If the player is currently at R , then a weapon and a character should be picked. If not, the agent should roll the dice and follow the path towards R , limited by the rolled number. If, after moving, the player is inside R , then the agent proceeds to pick a weapon and a character, otherwise, the turn ends.

Both weapons and characters are picked using the same criteria. As the agent tries to maximize the chance of acquiring new information, the only aspects taken into consideration are if the card was previously seen and who showed it. For each list $\{\text{ValidWeapons}, \text{ValidCharacters}\}$, it orders the cards descending by the rank of the player who showed them (according to Definition 3.3). Cards not yet seen receive rank (maximum player rank + 1). By such ordering, the priority of a card to be picked is first the cards not yet seen, then those seen from farthest to closest player. The agent just generates the suggestion using the picked room and the head of both ordered lists and informs it to the user.

After the suggestion is made, the new information acquired, if any, might allow the agent to make an accusation. So, it makes this test once again.

```

if can accuse &Person, &Room, &Weapon then
    | accuse Person, Room, Weapon;
else
    | Rooms ← find all pairs (Room, Path) from current position;
    | sort Rooms by Path length ascending;
    | if my position P is a room and not entered P this turn then
    |     | remove pair from Rooms containing P;
    | end
    | (Room, Path) ← first element of Rooms whose Room was not shown by next player or
    | the head of Rooms, if there's no such element;
    | if not currently in Room then
    |     | Steps ← roll dice;
    |     | walk Path limited by Steps;
    | end
    | if currently in Room then
    |     | Weapon ← first of valid weapons, sorted by position of player that showed it,
    |         | descending;
    |     | Person ← first of valid characters, sorted by position of player that showed it,
    |         | descending;
    |     | suggest Person, Room, Weapon;
    |     | if can accuse &Person, &Room, &Weapon then
    |         |     | accuse Person, Room, Weapon;
    |     | end
    | end
end

```

Figure 13: Pseudocode for the action picking process.

The predicate `char_rank/3` (Figure 14) applies Definition 3.3 to find a character's rank in relation to any other. Basically, given a reference character `MyChar`, it rotates the list of valid characters (which is already sorted in the order the characters must play their turns) to the left so `MyChar` becomes the first element. Then, the rank is returned as the position of the other char on the list. Notice that the predicate gives 0 for the rank of `MyChar` in relation to itself, and that if a character is not in the list, the maximum rank plus 1 (*i.e.*, the size of the list) is returned.

```

%%% Player rank.

%% pivot_split(List, Pivot, Left, Right) - Splits List in two parts:
%% - Left -> elements before Pivot
%% - Right -> Pivot plus the elements after it.
%% If Pivot is not in the list, Left = List and Right = [].
pivot_split([], _, [], []).
pivot_split([Pivot|Tail], Pivot, [], [Pivot|Tail]) :- !.
pivot_split([Head|Tail], Pivot, [Head|Left], Right) :-
    pivot_split(Tail, Pivot, Left, Right).

%% char_rank(MyChar, Char, Rank) - gives a Rank to Char relative to MyChar,
%% that's the order in which Char would play, after MyChar, starting from 0.
char_pos(_, [], Pos, Pos).
char_pos(Char, [Char|_], Pos, Pos) :- !.
char_pos(Char, [_|Tail], Cur, Pos) :-
    Next is Cur + 1,
    char_pos(Char, Tail, Next, Pos).
char_rank(MyChar, Char, Rank) :-
    valid_chars(ValidChars),
    pivot_split(ValidChars, MyChar, Left, Right),
    append(Right, Left, Rotated),
    char_pos(Char, Rotated, 0, Rank).

```

Figure 14: Finding a player rank in relation to another.

3.2.1 Picking a Room

Built-in predicate `findall` will be used on `closest_door` to generate a list of paths to all reachable rooms. If the agent is currently inside a room, a unitary path to it (containing just the exit cell) will be included in the list, as it should. Nonetheless, if the player was in this same room last turn, she is not allowed to use it now. The predicate `remove_if_not_entered/4` (Figure 15) deals with this case, generating a new list without any path to the current room, if necessary.

```

%% remove_if_not_entered(Rooms, MyChar, MyPos, Result) - if i'm in a room and did
%% not enter it this turn, remove it from the list.
remove_if_not_entered(Rooms, MyChar, MyPos, Result) :-
    valid_rooms(ValidRooms), member(MyPos, ValidRooms), % I'm in a room
    last_position(MyPos, MyChar), % I didn't enter it this turn
    delete(Rooms, (MyPos, _), Result), !.
remove_if_not_entered(Rooms, _, _, Rooms).

```

Figure 15: If the current room cannot be used, removes it.

Next, the agent uses predicate `check_secret_passage/3` (Figure 16), to check if there's a secret passage from the given position to a target room and, if so, add it to an existing list of paths.

```

%% check_secret_passage(Rooms, MyPos, Result) - if there's a secret passage
%% from position MyPos (a room) to a target room, adds (TargetRoom, 'secret passage')
%% to the beginning of Rooms (a list of (Room, Path)).
check_secret_passage(Rooms, MyPos, [(TargetRoom, 'secret passage')|Rooms]) :-
    passage(MyPos, TargetRoom), !.
check_secret_passage(Rooms, _, Rooms).

```

Figure 16: Including the secret passage on the list of paths, if there's one.

The list must then be sorted by path length (Figure 17). The relative order of the elements is determined by `room_compare`. If one of the compared elements is a secret passage, it will be ordered before any other, except when the other element is a unitary path (*i.e.*, contains

only the source position, thus requiring zero steps to enter a room). This happens when the agent is currently inside a room (or in the cell right next to a door, but the agent would always have entered a room by reaching such a cell, so that case should never happen).

```

%% room_compare(Order, A, B) - unifies Order to the relative order (<, >, =)
%% between two elements A and B of a room list (Room, Path).
%% a path in the form of a secret passage is greater than a unitary path, i.e.,
%% a path that take zero steps to reach the goal (only the source), and less
%% than any other path. all other paths are compared by length.
room_compare(>, (_, 'secret passage'), (_, [])) :- !.
room_compare(<, (_, 'secret passage'), _) :- !.
room_compare(Delta, (_, PathA), (_, PathB)) :-
    length(PathA, LenA), length(PathB, LenB),
    compare(Delta, LenA, LenB).

%% sort_rooms(Rooms, Sorted) - unifies the list of rooms, sorted by the
%% order defined on room_compare
sort_rooms(Rooms, Sorted) :- predsort(room_compare, Rooms, Sorted).

```

Figure 17: Sorting rooms by path length, including secret passages.

Finally, the last step is to take the first room in the sorted list that was not shown by the next player, or just the first room in the list if this condition cannot be met. Auxiliary predicate `pick_room_not_shown_next/5` is called by `pick_room/2` to solve this problem,, as shown in Figure 18.

```

%% pick_room_not_shown_next(MyChar, Rooms, SeenRooms, Room, Path) - given Rooms, a list of (Room, Path)
%% sorted by path, picks the first room that was not shown by the next player or the
%% first element in the list, if no such room exists.
pick_room_not_shown_next([], [], SeenRooms, Room, Path) :-
    append(_, [(Room, Path)], SeenRooms), !. % take the last element of SeenRooms
pick_room_not_shown_next(_, [(Room, Path)|_], _, Room, Path) :-
    \+ shown_room(_, Room), !.
pick_room_not_shown_next(MyChar, [(Room, Path)|_], _, Room, Path) :-
    shown_room(Char, Room),
    char_rank(MyChar, Char, Rank),
    Rank =\= 1, !.
pick_room_not_shown_next(MyChar, [(R, P)|Tail], SeenRooms, Room, Path) :-
    pick_room_not_shown_next(MyChar, Tail, [(R, P)|SeenRooms], Room, Path), !.

%% picks a suitable room to be used this turn (could be the room it's currently in)
pick_room(Room, Path) :-
    my_char(MyChar),
    position(MyPos, MyChar),
    findall((R, P), closest_door(MyPos, R, P), Rooms),
    remove_if_not_entered(Rooms, MyChar, MyPos, WithoutCurrent),
    check_secret_passage(WithoutCurrent, MyPos, WithPassage),
    sort_rooms(WithPassage, Sorted),
    pick_room_not_shown_next(MyChar, Sorted, [], Room, Path), !.

```

Figure 18: Top level predicate to pick a room.

4 The Interface