Design Document.

Testing a Multi-Agent Production System by Modeling Dodge Ball

For CGSC 4001 Special Topics in Cognitive Science
/CGSC 5001 Cognition and Artificial Systems

Authors:
Vincent Breault [VincentBreault@cmail.carleton.ca]
Drew Blackmore [DrewBlackmore@cmail.carleton.ca]

Carleton University
2014 April 18

This document provides a detailed explanation of the presented program, including all modules, the production system's inner workings and the simulation used to test said production system.

The program is an implementation of a production system tested on a simulation of a dodge ball game. The dodge ball game is a multi-agent game where players need to throw balls at the opposing team in order to send its members to jail. A team wins when all of the other team has been sent to jail. This simulation contains a variety of different actions, some aggressive, such as throw ball, some neutral, such as pick up ball, and some defensive, such as dodge ball. It also incorporates interesting elements of timing and accounts for the actions of many agents interacting in the same environment.

The production system is used to determine each agent's action at each time step – which production fires for each agent on every turn. This system finds the best action given the current situation and elects to use it to modify the environment.

In order for the simulation to work, the program needs an external database describing the initial state of the environment, which is provided in the package, under the name 'Database.txt'. This can be modified to add or remove statements from the initial state of the environment. Each of these statements in the database file follows the chunk3 grammar, that is: 'ThingX relation ThingY'. It is converted into chunks for processing.

The program itself is divided into modules for different functions, which will all be described in great detail later in this document.

The flow of the program is as follows:

- The database is imported and the environment is populated with facts in the form of chunk3s
- The agents are instantiated and populated with both a list of available productions containing vague referents to the agents themselves (such as 'CurrentPlayer'), other agents (such as 'OtherPlayer'), and a list of knowledge describing indirect consequences of actions
- The simulation loops through each agent. For each one:
    - Creates a list of modified productions for the specific instance with specific references to players (such as 'Player_1' instead of 'CurrentPlayer' and 'Player_2' instead of 'OtherPlayer')
    - Gathers the modified productions into a list of legal productions, which is determined by consulting both the current facts in the environment and the productions' left side statements
    - Applies any knowledge of indirect consequences
    - Ranks the list of legal productions by a point system depending on their desirability
    - Returns the production with the highest value
    - Modifies the environment according to that production's right side.
    - Checks to see if either team has won the game during this round
    - Applies pending actions

The productions have been written so that the reference to either players or ball is as broad as possible. For example, this means that when the production criteria or consequence is referencing the agent currently choosing which production to fire, the statement includes 'CurrentPlayer' and when it is considering other players, it includes 'OtherPlayer'. The same has been implemented for ball: 'AnyBall' and 'PlayerBall' for any ball and the current player's ball, respectively. This has been done so as to avoid dedicated productions for each possible instance of a production. Rather, the system has one large description covering, for instance, any other player the current player may want to throw a ball at. This necessitates that the productions be specified at the moment of consideration, meaning that from one broad production, several specific ones will be created, as described in the Interpreter Module section.

The interpreter itself, which makes the decision on which production need to be chosen, does so

using the following algorithm. It first goes through all the specified productions and keeps only the ones that are currently legal. Using a simple knowledge structure resembling productions, the agent deduces what indirect results come from using each production. These productions are then weighted depending on their effect on the state of the environment, giving more weight to productions that get the agent closer to its goal or gets the opposing team further from their own goal, and less weight to productions that take the agent further away from its goal. Once weighted, the production with the highest value is chosen for this agent.

The specific classes, methods and function of the modules and main program will be described below in each section. There are five modules (Chunk_Module, Game_Module, Import_Module, Interpreter_Module and Productions_Module) and one file for main program (Run_Program).

Chunk Module

The chunk module contains classes for the environment and the different types of chunks.

Class Environment

The environment class represents the environment in which the simulation is to be run. The environment is to contain all the facts needed for the interpreter to choose the productions. Its only attribute, instantiated through the method '__init__', is self.facts, which is a list of chunk3s describing all the aspects of the environment, such as ('Ball_1', 'Is_In', 'Red_Court'). The environment class also contains two methods:

- removeFacts(self, factsToBeRomoved): This method removes facts from the environment. It takes as input a specific fact, and if it finds it in the environment, it removes it.
- addFacts(self, factsToBeAdded): This method adds facts to the environment. It takes as input a specific fact, and if it can't find a duplicate of it in the environment, it adds it as a Chunk3.

Class chunk

The second class is the chunk class, representing a bit of memory. It has two attributes, an ID which identifies the chunk and an activation with a value between 0 and 1. This class has one method aside from '__init__':

- addToActivation(self, addend): adds the addend to the activation of the chunk and returns the new activation. Addend is a number between -1 and 1.

Class Chunk1

The third class is the chunk1 class, representing a bit of memory that has only one part. It creates a new instance of a chunk1, which is a single concept in memory. Its only method is:

- __init__(self, environment, ID, thingX=0.0, activation=0.0

The attributes of this class are :

- chunkType: is set to 1 as this is a chunk1
- ID: serves the same purpose as the ID in the class chunk
- thingX: is equal to ID if not changed
- activation: originally set to 0.0
- environment: is where this chunk is to be added.

Class Chunk3

The last class is the chunk3 class, which represents a fact that has 3 parts. Its only method is the '__init__' method. Its attributes are:

- chunkType: set to 3 as this is a chunk3

- ID : its identification
- thingX: is one of the two concepts of the fact
- relation: is a description of how thingX and thingY are related
- thingY: is the other concept
- activation: is a number between 0 and 1

And it is appended to the environment upon instantiation.

Game Module

The game module is used in order to instantiate game elements.

Although it was originally created to contain a wider variety of objects, it was simplified as to contain only the agent class.

Class Agent

The agent class is what creates the players of our multi-agent dodge ball simulation. There are no custom methods for the agents themselves. For each of the agents, the following attributes are given through the '__init__' built-in method in python:

- ID: String 'Player_' followed by a number. Used as identifier for a given player for actions that are specific to a single player, be it as performing the action or having an action performed to them.
- raceVar: String 'kitty'. Used to determine to which subset of the system's productions this agent has access. The current simulation uses only one type of agent but this allows for the possibility of a variety of agents with different sets of possible productions.
- listOfProductions: A list of all the productions to which the agent is entitled, which is determined at instantiation according to the 'raceVar' and the 'requirement' condition (see production module)
- knowledge: A list of production-like elements representing the agent's knowledge of the indirect consequences of certain actions.

Import Module

This module imports the information from the database and uses it to set the initial conditions of the simulation.

This module imports our modules Chunk, Game, Interpreter, and Productions, as well as the operator module.

Def importDatabase(filename, numberOfBalls, numberOfPlayers)

Its first function imports the database and uses it to generate the initial conditions of the game. In order for the database to generate players and balls, it must include at least the following two lines:

- Environment Contains Ball_{0}
- Environment Contains Player_{0}

The function opens up the specified Dodge Ball database, imports the information it contains, and generates the start condition of the game. In order to do this, it first imports the database through the open() function. For each line in the database, it populates the environment with a number of dodge balls, players and any other facts stated in the database by converting them into chunk3 objects, which automatically appends them to the environment along the way. During this process, it makes a list of all

the players and all the balls divides them into teams for the simulation and assigns them a position, on red court of on blue court. After that it also creates instances of the player class with each of the player in the list of player. The *{0}* elements in the database allows the user input to modify how many of this item is created. The system changes the *{0}* through a loop that runs a number of times defined by the user.

def createRedGoal(numberOfPlayers), createBlueGoal(numberOfPlayers)

The two other functions in this module work the same way. They are used to create the teams' goal states. The ideal goal state for any team is to have all of its players on their court and all of the opposing team in jail. These functions therefore create an instance of the environment class containing the chunk3s describing this ideal situation. The function only takes as arguments the number of players and returns an environment.

def returnListOfAgents():

Lastly, the module has a function returnListOfAgents which returns the list of agent created by the importDatabase function.

Interpreter Module

The interpreter module is where the general productions are transformed into specific productions, evaluated, and chosen to be performed by an agent.

Given that the general productions needs to be transformed into specific productions without changing the original list in the agents themselves, the module needs to import 'copy' from the python library. In order to facilitate the mathematical operations, we also import the module 'operator', again from python library.

def specifier(listOfThingToSpecify, agent, environment, playerTotal, ballCount):

This module is comprised of two functions. The first one takes the generalized productions and knowledge found in the Productions_Module and specifies them, turning statements like 'OtherPlayer' or 'AnyBall' into specific designators, such as 'Player_2' or 'Ball_3'. A sample simplified input for this function might be:

```
listOfThingsToSpecify = [ID=throwBall,
                         leftSideTrue = ['CurrentPlayer', 'Carrying', 'PlayerBall'],
                         leftSideFalse = ['CurrentPlayer', 'Is_In', 'Jail']
                                         ['OtherPlayer', 'Is_In', 'Jail']
                        rightSideAdd = ['PlayerBall', 'Flying_At', 'OtherPlayer']
                        rightSideDel = ['CurrentPlayer', 'Carrying', 'PlayerBall']]
agent = agent_1
environment = court
playerTotal = 2
ballCount = 2
```

If first loops through all productions or pieces of knowledge given to it, and specifies 'CurrentPlayer' in each attributes to the agent's ID. It also specifies 'PlayerBall' to the ball held by the agent given as an argument in the main function call and the 'CurrentPlayerCourt' and 'OtherPlayerCourt' are also specified to the agent's own or the opposing's team side of the court. In the sample production given in the sample input, this would leave the production as follows:

listOfThingsToSpecify = [ID=throwBall,
leftSideTrue = [‘Player_1’, ‘Carrying’, ‘Ball_1’],
leftSideFalse = [Player_1', 'Is_In', 'Jail']
['OtherPlayer', 'Is_In', 'Jail']
rightSideAdd = ['Ball_1', 'Flying_At', 'OtherPlayer']
rightSideDel = ['Player_1', 'Carrying', 'Ball_1']]

Once the agent specific values have been attributed to the list of production or knowledge, a second loop checks through all productions for either the words ‘OtherPlayer’ or ‘AnyBall’. These terms designate several possible values such as ‘Player_1’ and ‘Player_2’ or ‘Ball_1’ and ‘Ball_2’ for actions such as our sample throwBall. If one of these terms exists in the production’s attributes, it means it needs to be specified. When such change is needed, the function loops a number of time equal to the number of players, creating a new identical production each time and changing the ‘OtherPlayer’ value to all existing player ID with ‘Player_’ plus with a string of the iteration number and adds the number to the ID to differentiate it from similar productions. Again using the same sample production, with a playerCount of two, this would create two different productions:

listOfThingsToSpecify = [ID=throwBall1,
leftSideTrue = [‘Player_1’, ‘Carrying’, ‘Ball_1’],
leftSideFalse = [Player_1', 'Is_In', 'Jail']
['Player_1', 'Is_In', 'Jail']
rightSideAdd = ['Ball_1', 'Flying_At', 'Player_1']
rightSideDel = ['Player_1', 'Carrying', 'Ball_1']]

listOfThingsToSpecify = [ID=throwBall2,
leftSideTrue = [‘Player_1’, ‘Carrying’, ‘Ball_1’],
leftSideFalse = [Player_1', 'Is_In', 'Jail']
['Player_2', 'Is_In', 'Jail']
rightSideAdd = ['Ball_1', 'Flying_At', 'Player_2']
rightSideDel = ['Player_1', 'Carrying', 'Ball_1']]

Each new production is then added to a list of the variable production, which is returned at the end of the function.

def chooseProduction(agent, environment, myListOfGoals, enemyListOfGoals, playerTotal, ballCount):

The second and last function of the Interpreter module is the chooseProduction function. It first calls the specifier function, then creates a list of legal productions, and then ranks those productions through a point-assignment method. The function has a similar input to the specifier function:

- agent: the player running the production or making use of the knowledge.
- Environment: the environment which will be affected by the specified productions and against which the productions requirements are tested
- myListOfGoals: the ideal environment for the given agent
- enemyListOfGoals: the environment (ideal for the enemy) that the agent is attempting to avoid
- playerTotal: the number of agents in the game

- ballCount: is the number of balls in play

In order to avoid modifying the agent's list of productions, the function first makes a copy of the agent's production and knowledge list using the 'copy' module's 'deepcopy' function. These copies are then passed on to the specifier function in order to make all the agent specific productions that are needed to cover all possible actions depending on the current environment.

With this exhaustive list of all the specific actions, the function loops through all of them and creates a variable 'legal' with a value of 0 at the beginning of each iteration. It then loops through all of their left side attributes (leftSideTrue and leftSideFalse). For each statement that exists in both leftSideTrue (the list of conditions that must be true in the environment) and in the environment, the value of the 'legal' variable is increased by one. Likewise, for each statement that exists in both leftSideFalse (which is a list of facts that must not exist in the environment) and in the environment, the value of legal is reduced by one. If at the end of this cycle the value of legal is the same as the number of facts in leftSideTrue, we know that no false statement existed and that all true statements were found, meaning the production is legal. It is then added to a dictionary as a key with the value 0. The loop goes through all the productions in this way and populates the list called listOfLegalProductions, giving each of the production a value of 0.

Given that the consequences of some actions are indirect, they must be checked against a knowledge base for the agent to know their ultimate consequences. This modifies the productions, making their indirect causes direct. For example, 'throwBall' causes the ball you've thrown to go flying through the air at your target. The fact that this will put the target in jail is not explicitly stated in the results. The knowledge however states that a ball flying through the air at a target will result in the target being sent to jail. Thus, the knowledge would alter any production that caused a ball to go flying through the air to instead immediately cause its target to go to jail. This modification is what allows the agent to think ahead and choose productions that will move it closer to its goal state, even if not immediately. This is accomplished in the implementation by looping through every fact in the production's rightSideAdd list and changing any statement that matched a knowledge item's condition to said knowledge item's result. This means that our above sample would have its rightSideAdd value ['Ball_1', 'Flying_At', 'Player_2'] match the condition and therefore be changed to ['Player_2' 'Is_In' 'Jail'], which is in the agent's goal state.

This modification however is not a direct consequence of the action. This change must therefore not be kept in the production that is returned by the function which modifies the environment. To this end, before entering this last section of the code, the system created a copy of the variableProductionList called productionClone. The modifications take place in the variableProductionList but the initial production is still accessible through the clone.

Once modified, the variable production list is looped through one last time in order for the system to choose which production to use. This is done through point attribution. The system again loops through the modified variable production to assign point values to each production in the list of legal productions as modified by the knowledge base. This is done by comparing the rightSideAdd and rightSideDel value to the agent's goal state and the opposing team's goal state. Each time a statement is found matching a statement in a goal state, the value in the dictionary is modified in the following way:

1. Productions that add to the environment some fact in the agent's ideal goal state are worth +1 point for each such fact added.
2. Productions that add to the environment some fact in the enemy's ideal goal state are worth -1 point for each such fact added.
3. Productions that remove from the environment some fact in the enemy's ideal goal state are worth +1 point for each such fact added.

Once this is done, all the productions have their own value representing how much it advances their own goal and how much it hinders the other team. Considering that dictionaries are not ordered, if

two productions have the same value, they will be picked at random. Furthermore, in order to avoid doing nothing while another production is available, the production doNothing is given a value of -1 if it is not alone in the list.

Lastly, the function chooses the production with the highest value using the 'operator' build-in module's function max() and checks it's ID to find the cloned production that matches in order to have the correct direct consequence. This production from clonedProduction is then returned by the function.

Productions Module

The production module instantiates all the production objects in the program.

Class Production

The productions take the form of objects created by the class 'Production'. In order to facilitate the determination of whether a statement needs be true or false on the left side, or to be added to or removed from the environment on the right side, we made a design decision to give our productions four attributes each containing the relevant statements. No methods have been created for the productions, the only one being the '__init__' that initialises the object. The attributes are as follow:

- ID: A string with the production's identification, such as 'throwBall'. It is used in order to match specified productions to their instance during the weighting of productions in the interpreter.
- Requirement: A string 'kitty'. This string determines to which type(s) of agent the production is available.
- leftSideTrue: A list of facts that must be true in order for the production to be legal. Every fact on this list must exist in the environment in order for the production to be legal.
- leftSideFalse: A list of facts that cannot be true in order for the production to be legal. If a single statement on this list exists in the environment, the production cannot be legal.
- rightSideAdd: A list of facts that, when fired, the production adds to the environment. It tells the system which statements must be added if the production fires.
- rightSideDel: A list of facts that, when fired, the production removes from the environment. It tells the system which statements must be removed if the production fires.

def instantiateProduction():

This module also contains the function that creates all the general productions. These productions contain general terms to designate the current player considering the productions, the other players the productions can affect, and the balls that are either held by the player, flying through the air, or lying on the ground. The following table describes the productions our simulation used, all in their general states. Since we did not use any other value for requirement beside 'kitty' and the ID is equal to the name, these variables will be omitted from this table.

Table 1: List of the system's productions

throwBall	
leftSideTrue	['CurrentPlayer', 'Carrying', 'PlayerBall']
leftSideFalse	['CurrentPlayer', 'Is_In', 'Jail'] ['OtherPlayer', 'Is_In', 'Jail']
rightSideAdd	['PlayerBall', 'Flying_At', 'OtherPlayer']
rightSideDel	['CurrentPlayer', 'Carrying', 'PlayerBall']
pickUpBall	
leftSideTrue	['AnyBall', 'Is_In', 'CurrentPlayerCourt']

leftSideFalse	['CurrentPlayer', 'Carrying', 'PlayerBall'] ['CurrentPlayer', 'Is_In', 'Jail']
rightSideAdd	['CurrentPlayer', 'Carrying', 'AnyBall']
rightSideDel	['AnyBall', 'Is_In', 'CurrentPlayerCourt']
blockBall	
leftSideTrue	['AnyBall', 'Flying_At', 'CurrentPlayer'], ['CurrentPlayer', 'Carrying', 'PlayerBall']
leftSideFalse	['CurrentPlayer', 'Is_In', 'Jail']
rightSideAdd	['AnyBall', 'Is_In', 'CurrentPlayerCourt']
rightSideDel	['AnyBall', 'Flying_At', 'CurrentPlayer']
dodgeBall	
leftSideTrue	['AnyBall', 'Flying_At', 'CurrentPlayer']
leftSideFalse	['CurrentPlayer', 'Is_In', 'Jail']
rightSideAdd	['AnyBall', 'Is_In', 'CurrentPlayerCourt']
rightSideDel	['AnyBall', 'Flying_At', 'CurrentPlayer']
catchBall	
leftSideTrue	['AnyBall', 'Flying_At', 'CurrentPlayer']
leftSideFalse	['CurrentPlayer', 'Is_In', 'Jail'], ['CurrentPlayer', 'Carrying', 'PlayerBall']
rightSideAdd	['CurrentPlayer', 'Carrying', 'AnyBall']
rightSideDel	['AnyBall', 'Flying_At', 'CurrentPlayer']
doNothing	
leftSideTrue	[]
leftSideFalse	[]
rightSideAdd	[]
rightSideDel	[]

Class Knowledge

This class instantiates the agent's knowledge database. Each instance of the knowledge class represents a single piece of causal knowledge. In our system, causal knowledge takes the form of conditional statements. The condition is the antecedent. The result is the consequent. They are both rendered as three-part facts in a list, similar to Chunk3s. Therefore, any agent endowed with a piece of knowledge understands that any action that brings about the condition ultimately leads to the result. Just like the productions, this class holds no method of its own aside from the __init__. The attributes are as follow:

- ID: A string with the identification. Used for comparing the object with its specified counterpart.
- Condition: A three-part fact in a list. Used much like the left side of a production, describing what must be true for the result to occur.
- Result: A three-part fact in a list. Used much like the right side of a production, describing what results from a certain condition.

def instantiateKnowledge():

The knowledge objects were instantiated through this method. The two pieces of knowledge that were used in our system served to make the agent know the results of both throwing a ball at someone and of having a ball thrown at them (above and beyond balls flying through the air, that is). The following table describes our two knowledge objects, ID omitted:

Table 2: Knowledge database

getHit	
Condition	['AnyBall', 'Flying_At', 'CurrentPlayer']
Result	['CurrentPlayer', 'Is_In', 'Jail']
makeHit	
Condition	['PlayerBall', 'Flying_At', 'OtherPlayer']
Result	['OtherPlayer', 'Is_In', 'Jail']

Main Program

The main program is where the simulation happens. We decided that the throwBall action only puts players in jail if a ball thrown at them has been in the air for two iterations of the main loop. This allows agents to react to an incoming ball – the alternative would put them in jail instantly, leaving them no time to react. This way, an agent has time to make at least one decision, after a statement with a ball flying at them was added to the environment, before being put to jail.

The simulation is run through the main file, and in order to run the simulation, all the modules must first be imported, which is executed outside of a function. Afterwards, everything is done through the function 'main()'.

def main()

The first variable to be defined is the number of agents (or players) to be instantiated, called 'playerCount'. It is use-defined. Being a multi-agent system it is important for it to be able to perform with any number of agents. The second variable is also user defined, and is called 'ballCount', for the number of balls with which the agents can interact (or, thought of more generally, the number of items). The last of three user-defined variables is the name of the database to be used ('fileName'). Our system uses an external database containing all the facts in the environment, which, as discussed in the Import Module, populates the environment with chunks of information. These variables are used to create the initial state of the game.

The environment, defined in the variable 'court', gets populated through the import module's importDatabase function, with the fileName, ballCount and playerCount as arguments. The goals for the two teams are then instantiated through the import module with the function createRedGoal and createBlueGoal respectively, both using 'playerCount' as argument. These two variables are idealized environments, containing chunks of information that together describe a goal statement for a perfect 'court', depending on the player team. Lastly, a list of the agents, 'listOfAgents' gets populated with the agents created in the game module. With that initial state, agents having been given their productions and goal state as part of their own instantiation, the simulation is ready to begin.

The simulation is ran through a 'while' loop with the condition 'notDone == True', which changes to false as soon as one of the teams has all their players in jail. The first section of the loop, which is skipped over on the first iteration, checks for chunks in court describing an incoming ball that has been there for more than one turn. This will be described at the end of this section as it occurs only on the second iteration forward.

Each iteration, an action must be chosen for each of the players. An empty dictionary 'listOfActions' is therefore populated through a loop of the listOfAgent previously described. This is done by calling the interpreter module function 'chooseProduction' described in its respective section and passing it the following arguments:

- Players: An agent object from the list. This is the agent that will be performing the action and contains all the information relevant for this to occur, such as the list of production, the knowledge and its ID

- Court: A list of chunk3s representing all the information contained in the environment. This is so the agent may use the current state of the environment to know which actions are legal.
- redTeamGoal: A list of chunk3s representing all the information that an ideal environment would contain. This is used by the interpreter to compare evaluated productions with its ideal state (given that the current agent is on red team) for ranking.
- blueTeamGoal: A list of chunk3s representing all the information that an ideal environment would contain for the enemy team. This is used in the same way as 'redTeamGoal' in order for the interpreter to take into account enemy goals into ranking.
- playerTotal: Contains the current number of players. This is used by the 'specifier' function to create specific instances of each production depending on which agent is implied in said production.
- ballCount: Current number of balls. This is used by the 'specifier' function to create specific instances of each production depending on which ball is implied in said production.

The output of the function, a single production, is appended to the dictionary as key with the value 0 until the loop has gone through all agents.

Once the list is completed, all the actions are looped through to actuate the environment. This is done, for each action, by going through its 'rightSideAdd' and 'rightSideDel' attributes, and using the appropriate method from the Environment class, 'addFacts' and 'removeFacts' respectively, which take as arguments the current value of the attribute. Additionally, this loop makes sure that every 'throwBall' action is kept in the list by raising its value to 1. Any action with a value of 0 initially will modify the environment and if its value remains 0 at the end of the loop (which is to say, the action is not 'throwBall'), it will be removed from the dictionary.

At the end of the main loop, by comparing the number of players in jail for each team with the total number of players, the system checks for a winner. It declared a winner if an entire team is in jail.

Now on the second iteration, if the list of actions is not empty, the system will check for pending 'throwBall' actions to hit targets. If the value of said action is equal to one, it means it has just been thrown and therefore need not to hit right away. Its value will be increased by one again. On the other hand, if its value is equal to three (as it will be raised again on the next iteration), the system knows the target of the ball needs to be put in jail as it has not performed a defensive action (dodge, block or catch), and will change the environment 'court' to reflect that the player is in jail and that the ball is on that player's side of the court. This is done with the addFact and removeFact methods.

Through iterations, agents will pick up balls, throw them, dodge and block until all agents on one team are put to jail, at which point the program ends.

User Interface

The program has a text-based user interface. It asks the user for the input parameter necessary to run the program. During iterations, it prints to the screen interesting details such as player and ball locations and actions chosen by players. The following is an example of the program's initialization as well as the initial environment fact list and the first actions chosen by the agents and their results.

```
How many Kitties are playing per team?
There must be at least 1.
How many balls does each team receive?
There must be at least 1.
Please type in the full filename of the database you with to use (in quotes).
(For Purposes of the project, please input exactly the following: 'Database.txt')

Database.txt successfully opened.

('Environment', 'Contains', 'Red_Court')
('Environment', 'Contains', 'Blue_Court')
('Environment', 'Contains', 'Jail')
('Red_Team', 'Is_A', 'Team')
('Blue_Team', 'Is_A', 'Team')
('Environment', 'Contains', 'Ball_1')
('Environment', 'Contains', 'Ball_2')
('Environment', 'Contains', 'Player_1')
('Environment', 'Contains', 'Player_2')
('Environment', 'Contains', 'Player_3')
('Environment', 'Contains', 'Player_4')
('Ball_1', 'Is_In', 'Red_Court')
('Ball_2', 'Is_In', 'Blue_Court')
('Player_1', 'Is_On', 'Red_Team')
('Player_1', 'Is_In', 'Red_Court')
('Player_2', 'Is_On', 'Red_Team')
('Player_2', 'Is_In', 'Red_Court')
('Player_3', 'Is_On', 'Blue_Team')
('Player_3', 'Is_In', 'Blue_Court')
('Player_4', 'Is_On', 'Blue_Team')
('Player_4', 'Is_In', 'Blue_Court')


Player_1uses: pickUpBall111

('Ball_2', 'Is_In', 'Blue_Court')
('Player_1', 'Is_In', 'Red_Court')
('Player_2', 'Is_In', 'Red_Court')
('Player_3', 'Is_In', 'Blue_Court')
('Player_4', 'Is_In', 'Blue_Court')
('Player_1', 'Carrying', 'Ball_1')

Player_2uses: doNothing

('Ball_2', 'Is_In', 'Blue_Court')
('Player_1', 'Is_In', 'Red_Court')
('Player_2', 'Is_In', 'Red_Court')
('Player_3', 'Is_In', 'Blue_Court')
('Player_4', 'Is_In', 'Blue_Court')
('Player_1', 'Carrying', 'Ball_1')

Player_3uses: pickUpBall222
```