Sage Reference Manual: Game Theory

Release 8.8

The Sage Development Team

Jun 27, 2019

CONTENTS

CHAPTER

ONE

CO-OPERATIVE GAMES WITH FINITE PLAYERS

This module implements a class for a characteristic function cooperative game. Methods to calculate the Shapley value (a fair way of sharing common resources: see [**?**]) as well as test properties of the game (monotonicity, superadditivity) are also included.

AUTHORS:

- James Campbell and Vince Knight (06-2014): Original version

class sage.game_theory.cooperative_game.**CooperativeGame**(*characteristic_function*)
Bases: sage.structure.sage_object.SageObject

An object representing a co-operative game. Primarily used to compute the Shapley value, but can also provide other information.

INPUT:

- characteristic_function – a dictionary containing all possible sets of players:
  - key - each set must be entered as a tuple.
  - value - a real number representing each set of players contribution

EXAMPLES:

The type of game that is currently implemented is referred to as a Characteristic function game. This is a game on a set of players Ω that is defined by a value function $v : C \to \mathbf{R}$ where $C = 2^{\Omega}$ is the set of all coalitions of players. Let $N := |\Omega|$. An example of such a game is shown below:

$$v(c) = \begin{cases} 0 & \text{if } c = \emptyset, \\ 6 & \text{if } c = \{1\}, \\ 12 & \text{if } c = \{2\}, \\ 42 & \text{if } c = \{3\}, \\ 12 & \text{if } c = \{1, 2\}, \\ 42 & \text{if } c = \{1, 3\}, \\ 42 & \text{if } c = \{2, 3\}, \\ 42 & \text{if } c = \{1, 2, 3\}. \end{cases}$$

The function v can be thought of as a record of contribution of individuals and coalitions of individuals. Of interest, becomes how to fairly share the value of the grand coalition (Ω)? This class allows for such an answer to be formulated by calculating the Shapley value of the game.

Basic examples of how to implement a co-operative game. These functions will be used repeatedly in other examples.

```
sage: integer_function = {(): 0,
....:                     (1,): 6,
....:                     (2,): 12,
....:                     (3,): 42,
....:                     (1, 2,): 12,
....:                     (1, 3,): 42,
....:                     (2, 3,): 42,
....:                     (1, 2, 3,): 42}
sage: integer_game = CooperativeGame(integer_function)
```

We can also use strings instead of numbers.

```
sage: letter_function = {(): 0,
....:                    ('A',): 6,
....:                    ('B',): 12,
....:                    ('C',): 42,
....:                    ('A', 'B',): 12,
....:                    ('A', 'C',): 42,
....:                    ('B', 'C',): 42,
....:                    ('A', 'B', 'C',): 42}
sage: letter_game = CooperativeGame(letter_function)
```

Please note that keys should be tuples. `'1, 2, 3'` is not a valid key, neither is `123`. The correct input would be `(1, 2, 3)`. Similarly, for coalitions containing a single element the bracket notation (which tells Sage that it is a tuple) must be used. So `(1)`, `(1,)` are correct however simply inputting 1 is not.

Characteristic function games can be of various types.

A characteristic function game $G = (N, v)$ is monotone if it satisfies $v(C_2) \geq v(C_1)$ for all $C_1 \subseteq C_2$. A characteristic function game $G = (N, v)$ is superadditive if it satisfies $v(C_1 \cup C_2) \geq v(C_1) + v(C_2)$ for all $C_1, C_2 \subseteq 2^{\Omega}$ such that $C_1 \cap C_2 = \emptyset$.

We can test if a game is monotonic or superadditive.

```
sage: letter_game.is_monotone()
True
sage: letter_game.is_superadditive()
False
```

Instances have a basic representation that will display basic information about the game:

```
sage: letter_game
A 3 player co-operative game
```

It can be shown that the "fair" payoff vector, referred to as the Shapley value is given by the following formula:

$$\phi_i(G) = \frac{1}{N!} \sum_{\pi \in \Pi_n} \Delta_{\pi}^{G}(i),$$

where the summation is over the permutations of the players and the marginal contributions of a player for a given permutation is given as:

$$\Delta_{\pi}^{G}(i) = v\big(S_{\pi}(i) \cup \{i\}\big) - v\big(S_{\pi}(i)\big)$$

where $S_{\pi}(i)$ is the set of predecessors of i in π, i.e. $S_{\pi}(i) = \{j \mid \pi(i) > \pi(j)\}$ (or the number of inversions of the form (i, j)).

This payoff vector is "fair" in that it has a collection of properties referred to as: efficiency, symmetry, additivity and Null player. Some of these properties are considered in this documentation (and tests are implemented in the class) but for a good overview see [**?**].

Note ([**?**]) that an equivalent formula for the Shapley value is given by:

$$\phi_i(G) = \sum_{S \subseteq \Omega} \sum_{p \in S} \frac{(|S|-1)!(N-|S|)!}{N!}\big(v(S) - v(S \setminus \{p\})\big) = \sum_{S \subseteq \Omega} \sum_{p \in S} \frac{1}{|S|\binom{N}{|S|}}\big(v(S) - v(S \setminus \{p\})\big).$$

This later formulation is implemented in Sage and requires $2^N - 1$ calculations instead of $N!$.

To compute the Shapley value in Sage is simple:

```
sage: letter_game.shapley_value()
{'A': 2, 'B': 5, 'C': 35}
```

The following example implements a (trivial) 10 player characteristic function game with $v(c) = |c|$ for all $c \in 2^\Omega$.

```
sage: def simple_characteristic_function(N):
....:     return {tuple(coalition) : len(coalition)
....:                     for coalition in subsets(range(N))}
sage: g = CooperativeGame(simple_characteristic_function(10))
sage: g.shapley_value()
{0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1}
```

For very large games it might be worth taking advantage of the particular problem structure to calculate the Shapley value and there are also various approximation approaches to obtaining the Shapley value of a game (see [**?**] for one such example). Implementing these would be a worthwhile development For more information about the computational complexity of calculating the Shapley value see [**?**].

We can test 3 basic properties of any payoff vector λ. The Shapley value (described above) is known to be the unique payoff vector that satisfies these and 1 other property not implemented here (additivity). They are:

- Efficiency - $\sum_{i=1}^{N} \lambda_i = v(\Omega)$ In other words, no value of the total coalition is lost.
- The nullplayer property - If there exists an i such that $v(C \cup i) = v(C)$ for all $C \in 2^\Omega$ then, $\lambda_i = 0$. In other words: if a player does not contribute to any coalition then that player should receive no payoff.
- Symmetry property - If $v(C \cup i) = v(C \cup j)$ for all $C \in 2^\Omega \setminus \{i, j\}$, then $x_i = x_j$. If players contribute symmetrically then they should get the same payoff:

```
sage: payoff_vector = letter_game.shapley_value()
sage: letter_game.is_efficient(payoff_vector)
True
sage: letter_game.nullplayer(payoff_vector)
True
sage: letter_game.is_symmetric(payoff_vector)
True
```

Any payoff vector can be passed to the game and these properties can once again be tested:

```
sage: payoff_vector = {'A': 0, 'C': 35, 'B': 3}
sage: letter_game.is_efficient(payoff_vector)
False
sage: letter_game.nullplayer(payoff_vector)
True
sage: letter_game.is_symmetric(payoff_vector)
True
```

is_efficient (*payoff_vector*)

Return `True` if `payoff_vector` is efficient.

A payoff vector v is efficient if $\sum_{i=1}^{N} \lambda_i = v(\Omega)$; in other words, no value of the total coalition is lost.

INPUT:

- `payoff_vector` – a dictionary where the key is the player and the value is their payoff

EXAMPLES:

An efficient payoff vector:

```
sage: letter_function = {(): 0,
....:                    ('A',): 6,
....:                    ('B',): 12,
....:                    ('C',): 42,
....:                    ('A', 'B',): 12,
....:                    ('A', 'C',): 42,
....:                    ('B', 'C',): 42,
....:                    ('A', 'B', 'C',): 42}
sage: letter_game = CooperativeGame(letter_function)
sage: letter_game.is_efficient({'A': 14, 'B': 14, 'C': 14})
True

sage: letter_function = {(): 0,
....:                    ('A',): 6,
....:                    ('B',): 12,
....:                    ('C',): 42,
....:                    ('A', 'B',): 12,
....:                    ('A', 'C',): 42,
....:                    ('B', 'C',): 42,
....:                    ('A', 'B', 'C',): 42}
sage: letter_game = CooperativeGame(letter_function)
sage: letter_game.is_efficient({'A': 10, 'B': 14, 'C': 14})
False
```

A longer example:

```
sage: long_function = {(): 0,
....:                  (1,): 0,
....:                  (2,): 0,
....:                  (3,): 0,
....:                  (4,): 0,
....:                  (1, 2): 0,
....:                  (1, 3): 0,
....:                  (1, 4): 0,
....:                  (2, 3): 0,
....:                  (2, 4): 0,
....:                  (3, 4): 0,
....:                  (1, 2, 3): 0,
....:                  (1, 2, 4): 45,
....:                  (1, 3, 4): 40,
....:                  (2, 3, 4): 0,
....:                  (1, 2, 3, 4): 65}
sage: long_game = CooperativeGame(long_function)
sage: long_game.is_efficient({1: 20, 2: 20, 3: 5, 4: 20})
True
```

is_monotone()

Return `True` if `self` is monotonic.

A game $G = (N, v)$ is monotonic if it satisfies $v(C_2) \geq v(C_1)$ for all $C_1 \subseteq C_2$.

EXAMPLES:

A simple game that is monotone:

```
sage: integer_function = {(): 0,
....:                     (1,): 6,
....:                     (2,): 12,
....:                     (3,): 42,
....:                     (1, 2,): 12,
....:                     (1, 3,): 42,
....:                     (2, 3,): 42,
....:                     (1, 2, 3,): 42}
sage: integer_game = CooperativeGame(integer_function)
sage: integer_game.is_monotone()
True
```

An example when the game is not monotone:

```
sage: integer_function = {(): 0,
....:                     (1,): 6,
....:                     (2,): 12,
....:                     (3,): 42,
....:                     (1, 2,): 10,
....:                     (1, 3,): 42,
....:                     (2, 3,): 42,
....:                     (1, 2, 3,): 42}
sage: integer_game = CooperativeGame(integer_function)
sage: integer_game.is_monotone()
False
```

An example on a longer game:

```
sage: long_function = {(): 0,
....:                  (1,): 0,
....:                  (2,): 0,
....:                  (3,): 0,
....:                  (4,): 0,
....:                  (1, 2): 0,
....:                  (1, 3): 0,
....:                  (1, 4): 0,
....:                  (2, 3): 0,
....:                  (2, 4): 0,
....:                  (3, 4): 0,
....:                  (1, 2, 3): 0,
....:                  (1, 2, 4): 45,
....:                  (1, 3, 4): 40,
....:                  (2, 3, 4): 0,
....:                  (1, 2, 3, 4): 65}
sage: long_game = CooperativeGame(long_function)
sage: long_game.is_monotone()
True
```

is_superadditive()

Return `True` if `self` is superadditive.

A characteristic function game $G = (N, v)$ is superadditive if it satisfies $v(C_1 \cup C_2) \geq v(C_1) + v(C_2)$ for all $C_1, C_2 \subseteq 2^{\Omega}$ such that $C_1 \cap C_2 = \emptyset$.

EXAMPLES:

An example that is not superadditive:

```
sage: integer_function = {(): 0,
....:                     (1,): 6,
....:                     (2,): 12,
....:                     (3,): 42,
....:                     (1, 2,): 12,
....:                     (1, 3,): 42,
....:                     (2, 3,): 42,
....:                     (1, 2, 3,): 42}
sage: integer_game = CooperativeGame(integer_function)
sage: integer_game.is_superadditive()
False
```

An example that is superadditive:

```
sage: A_function = {(): 0,
....:                (1,): 6,
....:                (2,): 12,
....:                (3,): 42,
....:                (1, 2,): 18,
....:                (1, 3,): 48,
....:                (2, 3,): 55,
....:                (1, 2, 3,): 80}
sage: A_game = CooperativeGame(A_function)
sage: A_game.is_superadditive()
True
```

An example with a longer game that is superadditive:

```
sage: long_function = {(): 0,
....:                  (1,): 0,
....:                  (2,): 0,
....:                  (3,): 0,
....:                  (4,): 0,
....:                  (1, 2): 0,
....:                  (1, 3): 0,
....:                  (1, 4): 0,
....:                  (2, 3): 0,
....:                  (2, 4): 0,
....:                  (3, 4): 0,
....:                  (1, 2, 3): 0,
....:                  (1, 2, 4): 45,
....:                  (1, 3, 4): 40,
....:                  (2, 3, 4): 0,
....:                  (1, 2, 3, 4): 65}
sage: long_game = CooperativeGame(long_function)
sage: long_game.is_superadditive()
True
```

An example with a longer game that is not:

```
sage: long_function = {(): 0,
....:                  (1,): 0,
....:                  (2,): 0,
....:                  (3,): 55,
....:                  (4,): 0,
....:                  (1, 2): 0,
....:                  (1, 3): 0,
```

(continues on next page)

(continued from previous page)

```
....:                     (1, 4): 0,
....:                     (2, 3): 0,
....:                     (2, 4): 0,
....:                     (3, 4): 0,
....:                     (1, 2, 3): 0,
....:                     (1, 2, 4): 45,
....:                     (1, 3, 4): 40,
....:                     (2, 3, 4): 0,
....:                     (1, 2, 3, 4): 85}
sage: long_game = CooperativeGame(long_function)
sage: long_game.is_superadditive()
False
```

is_symmetric(*payoff_vector*)

Return `True` if `payoff_vector` possesses the symmetry property.

A payoff vector possesses the symmetry property if $v(C \cup i) = v(C \cup j)$ for all $C \in 2^{\Omega} \setminus \{i, j\}$, then $x_i = x_j$.

INPUT:

- `payoff_vector` – a dictionary where the key is the player and the value is their payoff

EXAMPLES:

A payoff vector that has the symmetry property:

```
sage: letter_function = {(): 0,
....:                    ('A',): 6,
....:                    ('B',): 12,
....:                    ('C',): 42,
....:                    ('A', 'B',): 12,
....:                    ('A', 'C',): 42,
....:                    ('B', 'C',): 42,
....:                    ('A', 'B', 'C',): 42}
sage: letter_game = CooperativeGame(letter_function)
sage: letter_game.is_symmetric({'A': 5, 'B': 14, 'C': 20})
True
```

A payoff vector that returns `False`:

```
sage: integer_function = {(): 0,
....:                     (1,): 12,
....:                     (2,): 12,
....:                     (3,): 42,
....:                     (1, 2,): 12,
....:                     (1, 3,): 42,
....:                     (2, 3,): 42,
....:                     (1, 2, 3,): 42}
sage: integer_game = CooperativeGame(integer_function)
sage: integer_game.is_symmetric({1: 2, 2: 5, 3: 35})
False
```

A longer example for symmetry:

```
sage: long_function = {(): 0,
....:                  (1,): 0,
....:                  (2,): 0,
```

(continues on next page)

(continued from previous page)

```
....:                     (3,): 0,
....:                     (4,): 0,
....:                     (1, 2): 0,
....:                     (1, 3): 0,
....:                     (1, 4): 0,
....:                     (2, 3): 0,
....:                     (2, 4): 0,
....:                     (3, 4): 0,
....:                     (1, 2, 3): 0,
....:                     (1, 2, 4): 45,
....:                     (1, 3, 4): 40,
....:                     (2, 3, 4): 0,
....:                     (1, 2, 3, 4): 65}
sage: long_game = CooperativeGame(long_function)
sage: long_game.is_symmetric({1: 20, 2: 20, 3: 5, 4: 20})
True
```

nullplayer(*payoff_vector*)

Return `True` if `payoff_vector` possesses the nullplayer property.

A payoff vector v has the nullplayer property if there exists an i such that $v(C \cup i) = v(C)$ for all $C \in 2^{\Omega}$ then, $\lambda_i = 0$. In other words: if a player does not contribute to any coalition then that player should receive no payoff.

INPUT:

- `payoff_vector` – a dictionary where the key is the player and the value is their payoff

EXAMPLES:

A payoff vector that returns `True`:

```
sage: letter_function = {(): 0,
....:                    ('A',): 0,
....:                    ('B',): 12,
....:                    ('C',): 42,
....:                    ('A', 'B',): 12,
....:                    ('A', 'C',): 42,
....:                    ('B', 'C',): 42,
....:                    ('A', 'B', 'C',): 42}
sage: letter_game = CooperativeGame(letter_function)
sage: letter_game.nullplayer({'A': 0, 'B': 14, 'C': 14})
True
```

A payoff vector that returns `False`:

```
sage: A_function = {(): 0,
....:               (1,): 0,
....:               (2,): 12,
....:               (3,): 42,
....:               (1, 2,): 12,
....:               (1, 3,): 42,
....:               (2, 3,): 55,
....:               (1, 2, 3,): 55}
sage: A_game = CooperativeGame(A_function)
sage: A_game.nullplayer({1: 10, 2: 10, 3: 25})
False
```

A longer example for nullplayer:

```
sage: long_function = {(): 0,
....:                  (1,): 0,
....:                  (2,): 0,
....:                  (3,): 0,
....:                  (4,): 0,
....:                  (1, 2): 0,
....:                  (1, 3): 0,
....:                  (1, 4): 0,
....:                  (2, 3): 0,
....:                  (2, 4): 0,
....:                  (3, 4): 0,
....:                  (1, 2, 3): 0,
....:                  (1, 2, 4): 45,
....:                  (1, 3, 4): 40,
....:                  (2, 3, 4): 0,
....:                  (1, 2, 3, 4): 65}
sage: long_game = CooperativeGame(long_function)
sage: long_game.nullplayer({1: 20, 2: 20, 3: 5, 4: 20})
True
```

shapley_value()

Return the Shapley value for `self`.

The Shapley value is the "fair" payoff vector and is computed by the following formula:

$$\phi_i(G) = \sum_{S \subseteq \Omega} \sum_{p \in S} \frac{1}{|S| \binom{N}{|S|}} \big(v(S) - v(S \setminus \{p\})\big).$$

EXAMPLES:

A typical example of computing the Shapley value:

```
sage: integer_function = {(): 0,
....:                     (1,): 6,
....:                     (2,): 12,
....:                     (3,): 42,
....:                     (1, 2,): 12,
....:                     (1, 3,): 42,
....:                     (2, 3,): 42,
....:                     (1, 2, 3,): 42}
sage: integer_game = CooperativeGame(integer_function)
sage: integer_game.player_list
(1, 2, 3)
sage: integer_game.shapley_value()
{1: 2, 2: 5, 3: 35}
```

A longer example of the Shapley value:

```
sage: long_function = {(): 0,
....:                  (1,): 0,
....:                  (2,): 0,
....:                  (3,): 0,
....:                  (4,): 0,
....:                  (1, 2): 0,
....:                  (1, 3): 0,
....:                  (1, 4): 0,
....:                  (2, 3): 0,
....:                  (2, 4): 0,
```

(continues on next page)

(continued from previous page)

```
....:                    (3, 4): 0,
....:                    (1, 2, 3): 0,
....:                    (1, 2, 4): 45,
....:                    (1, 3, 4): 40,
....:                    (2, 3, 4): 0,
....:                    (1, 2, 3, 4): 65}
sage: long_game = CooperativeGame(long_function)
sage: long_game.shapley_value()
{1: 70/3, 2: 10, 3: 25/3, 4: 70/3}
```

CHAPTER

TWO

MATCHING GAMES.

This module implements a class for matching games (stable marriage problems) [**?**]. At present the extended Gale-Shapley algorithm is implemented which can be used to obtain stable matchings.

AUTHORS:

- James Campbell and Vince Knight 06-2014: Original version

class sage.game_theory.matching_game.**MatchingGame**(*generator*, *revr=None*)

Bases: sage.structure.sage_object.SageObject

A matching game.

A matching game (also called a stable matching problem) models a situation in a population of N suitors and N reviewers. Suitors and reviewers rank their preferences and attempt to find a match.

Formally, a matching game of size N is defined by two disjoint sets S and R of size N. Associated to each element of S and R is a preference list:

$$f : S \to R^N \text{ and } g : R \to S^N.$$

Here is an example of matching game on 4 players:

$$S = \{J, K, L, M\},$$
$$R = \{A, B, C, D\}.$$

With preference functions:

$$f(s) = \begin{cases} (A, D, C, B) & \text{if } s = J, \\ (A, B, C, D) & \text{if } s = K, \\ (B, D, C, A) & \text{if } s = L, \\ (C, A, B, D) & \text{if } s = M, \end{cases}$$

$$g(s) = \begin{cases} (L, J, K, M) & \text{if } s = A, \\ (J, M, L, K) & \text{if } s = B, \\ (K, M, L, J) & \text{if } s = C, \\ (M, K, J, L) & \text{if } s = D. \end{cases}$$

INPUT:

Two potential inputs are accepted (see below to see the effect of each):

- `reviewer/suitors_preferences` – a dictionary containing the preferences of all players:
  - key - each reviewer/suitors
  - value - a tuple of suitors/reviewers

OR:

- `integer` – an integer simply representing the number of reviewers and suitors.

To implement the above game in Sage:

```
sage: suitr_pref = {'J': ('A', 'D', 'C', 'B'),
....:               'K': ('A', 'B', 'C', 'D'),
....:               'L': ('B', 'D', 'C', 'A'),
....:               'M': ('C', 'A', 'B', 'D')}
sage: reviewr_pref = {'A': ('L', 'J', 'K', 'M'),
....:                 'B': ('J', 'M', 'L', 'K'),
....:                 'C': ('K', 'M', 'L', 'J'),
....:                 'D': ('M', 'K', 'J', 'L')}
sage: m = MatchingGame([suitr_pref, reviewr_pref])
sage: m
A matching game with 4 suitors and 4 reviewers
sage: m.suitors()
('J', 'K', 'L', 'M')
sage: m.reviewers()
('A', 'B', 'C', 'D')
```

A matching M is any bijection between S and R. If $s \in S$ and $r \in R$ are matched by M we denote:

$$M(s) = r.$$

On any given matching game, one intends to find a matching that is stable. In other words, so that no one individual has an incentive to break their current match.

Formally, a stable matching is a matching that has no blocking pairs. A blocking pair is any pair (s, r) such that $M(s) \neq r$ but s prefers r to $M(r)$ and r prefers s to $M^{-1}(r)$.

To obtain the stable matching in Sage we use the `solve` method which uses the extended Gale-Shapley algorithm [**?**]:

```
sage: m.solve()
{'J': 'A', 'K': 'C', 'L': 'D', 'M': 'B'}
```

Matchings have a natural representations as bipartite graphs:

```
sage: plot(m)
Graphics object consisting of 13 graphics primitives
```

The above plots the bipartite graph associated with the matching. This plot can be accessed directly:

```
sage: graph = m.bipartite_graph()
sage: graph
Bipartite graph on 8 vertices
```

It is possible to initiate a matching game without having to name each suitor and reviewer:

```
sage: n = 8
sage: big_game = MatchingGame(n)
sage: big_game.suitors()
(1, 2, 3, 4, 5, 6, 7, 8)
sage: big_game.reviewers()
(-1, -2, -3, -4, -5, -6, -7, -8)
```

If we attempt to obtain the stable matching for the above game, without defining the preference function we obtain an error:

```
sage: big_game.solve()
Traceback (most recent call last):
...
ValueError: suitor preferences are not complete
```

To continue we have to populate the preference dictionary. Here is one example where the preferences are simply the corresponding element of the permutation group:

```
sage: from itertools import permutations
sage: suitr_preferences = list(permutations([-i-1 for i in range(n)]))
sage: revr_preferences = list(permutations([i+1 for i in range(n)]))
sage: for player in range(n):
....:     big_game.suitors()[player].pref = suitr_preferences[player]
....:     big_game.reviewers()[player].pref = revr_preferences[-player]
sage: big_game.solve()
{1: -1, 2: -8, 3: -6, 4: -7, 5: -5, 6: -4, 7: -3, 8: -2}
```

Note that we can also combine the two ways of creating a game. For example here is an initial matching game:

```
sage: suitrs = {'Romeo': ('Juliet', 'Rosaline'),
....:           'Mercutio': ('Juliet', 'Rosaline')}
sage: revwrs = {'Juliet': ('Romeo', 'Mercutio'),
....:           'Rosaline': ('Mercutio', 'Romeo')}
sage: g = MatchingGame(suitrs, revwrs)
```

Let us assume that all of a sudden a new pair of suitors and reviewers is added but their names are not known:

```
sage: g.add_reviewer()
sage: g.add_suitor()
sage: g.reviewers()
(-3, 'Juliet', 'Rosaline')
sage: g.suitors()
(3, 'Mercutio', 'Romeo')
```

Note that when adding a reviewer or a suitor all preferences are wiped:

```
sage: [s.pref for s in g.suitors()]
[[], [], []]
sage: [r.pref for r in g.reviewers()]
[[], [], []]
```

If we now try to solve the game we will get an error as we have not specified the preferences which will need to be updated:

```
sage: g.solve()
Traceback (most recent call last):
...
ValueError: suitor preferences are not complete
```

Here we update the preferences so that the new reviewers and suitors do not affect things too much (they prefer each other and are the least preferred of the others):

```
sage: g.suitors()[1].pref = suitrs['Mercutio'] + (-3,)
sage: g.suitors()[2].pref = suitrs['Romeo'] + (-3,)
sage: g.suitors()[0].pref = (-3, 'Juliet', 'Rosaline')
sage: g.reviewers()[2].pref = revwrs['Rosaline'] + (3,)
```

(continues on next page)

(continued from previous page)

```
sage: g.reviewers()[1].pref = revwrs['Juliet'] + (3,)
sage: g.reviewers()[0].pref = (3, 'Romeo', 'Mercutio')
```

Now the game can be solved:

```
sage: D = g.solve()
sage: D['Mercutio']
'Rosaline'
sage: D['Romeo']
'Juliet'
sage: D[3]
-3
```

Note that the above could be equivalently (and more simply) carried out by simply updated the original preference dictionaries:

```
sage: for key in suitrs:
....:     suitrs[key] = suitrs[key] + (-3,)
sage: for key in revwrs:
....:     revwrs[key] = revwrs[key] + (3,)
sage: suitrs[3] = (-3, 'Juliet', 'Rosaline')
sage: revwrs[-3] = (3, 'Romeo', 'Mercutio')
sage: g = MatchingGame(suitrs, revwrs)
sage: D = g.solve()
sage: D['Mercutio']
'Rosaline'
sage: D['Romeo']
'Juliet'
sage: D[3]
-3
```

It can be shown that the Gale-Shapley algorithm will return the stable matching that is optimal from the point of view of the suitors and is in fact the worst possible matching from the point of view of the reviewers. To quickly obtain the matching that is optimal for the reviewers we use the `solve` method with the `invert=True` option:

```
sage: left_dict = {'a': ('A', 'B', 'C'),
....:              'b': ('B', 'C', 'A'),
....:              'c': ('B', 'A', 'C')}
sage: right_dict = {'A': ('b', 'c', 'a'),
....:               'B': ('a', 'c', 'b'),
....:               'C': ('a', 'b', 'c')}
sage: quick_game = MatchingGame([left_dict, right_dict])
sage: quick_game.solve()
{'a': 'A', 'b': 'C', 'c': 'B'}
sage: quick_game.solve(invert=True)
{'A': 'c', 'B': 'a', 'C': 'b'}
```

EXAMPLES:

8 player letter game:

```
sage: suitr_pref = {'J': ('A', 'D', 'C', 'B'),
....:               'K': ('A', 'B', 'C', 'D'),
....:               'L': ('B', 'D', 'C', 'A'),
....:               'M': ('C', 'A', 'B', 'D')}
sage: reviewr_pref = {'A': ('L', 'J', 'K', 'M'),
....:                 'B': ('J', 'M', 'L', 'K'),
```

(continues on next page)

(continued from previous page)

```
....:                 'C': ('K', 'M', 'L', 'J'),
....:                 'D': ('M', 'K', 'J', 'L')}
sage: m = MatchingGame([suitr_pref, reviewr_pref])
sage: m.suitors()
('J', 'K', 'L', 'M')
sage: m.reviewers()
('A', 'B', 'C', 'D')
```

Also works for numbers:

```
sage: suit = {0: (3, 4),
....:         1: (3, 4)}
sage: revr = {3: (0, 1),
....:         4: (1, 0)}
sage: g = MatchingGame([suit, revr])
```

Can create a game from an integer. This gives default set of preference functions:

```
sage: g = MatchingGame(3)
sage: g
A matching game with 3 suitors and 3 reviewers
```

We have an empty set of preferences for a default named set of preferences:

```
sage: for s in g.suitors():
....:     s, s.pref
(1, [])
(2, [])
(3, [])
sage: for r in g.reviewers():
....:     r, r.pref
(-1, [])
(-2, [])
(-3, [])
```

Before trying to solve such a game the algorithm will check if it is complete or not:

```
sage: g.solve()
Traceback (most recent call last):
...
ValueError: suitor preferences are not complete
```

To be able to obtain the stable matching we must input the preferences:

```
sage: for s in g.suitors():
....:   s.pref = (-1, -2, -3)
sage: for r in g.reviewers():
....:   r.pref = (1, 2, 3)
sage: g.solve()
{1: -1, 2: -2, 3: -3}
```

add_reviewer(*name=None*)

Add a reviewer to the game.

INPUT:

- `name` – can be a string or number; if left blank will automatically generate an integer

EXAMPLES:

Creating a two player game:

```
sage: g = MatchingGame(2)
sage: g.reviewers()
(-1, -2)
```

Adding a suitor without specifying a name:

```
sage: g.add_reviewer()
sage: g.reviewers()
(-1, -2, -3)
```

Adding a suitor while specifying a name:

```
sage: g.add_reviewer(10)
sage: g.reviewers()
(-1, -2, -3, 10)
```

Note that now our game is no longer complete:

```
sage: g._is_complete()
Traceback (most recent call last):
...
ValueError: must have the same number of reviewers as suitors
```

Note that an error is raised if one tries to add a reviewer with a name that already exists:

```
sage: g.add_reviewer(10)
Traceback (most recent call last):
...
ValueError: a reviewer with name "10" already exists
```

If we add a reviewer without passing a name then the name of the reviewer will not use one that is already chosen:

```
sage: suit = {0: (-1,  -3),
....:         1: (-3, -1)}
sage: revr = {-1: (0, 1),
....:         -3: (1, 0)}
sage: g = MatchingGame([suit, revr])
sage: g.reviewers()
(-1, -3)

sage: g.add_reviewer()
sage: g.reviewers()
(-1, -3, -4)
```

add_suitor(*name=None*)

Add a suitor to the game.

INPUT:

- `name` – can be a string or a number; if left blank will automatically generate an integer

EXAMPLES:

Creating a two player game:

```
sage: g = MatchingGame(2)
sage: g.suitors()
(1, 2)
```

Adding a suitor without specifying a name:

```
sage: g.add_suitor()
sage: g.suitors()
(1, 2, 3)
```

Adding a suitor while specifying a name:

```
sage: g.add_suitor('D')
sage: g.suitors()
(1, 2, 3, 'D')
```

Note that now our game is no longer complete:

```
sage: g._is_complete()
Traceback (most recent call last):
...
ValueError: must have the same number of reviewers as suitors
```

Note that an error is raised if one tries to add a suitor with a name that already exists:

```
sage: g.add_suitor('D')
Traceback (most recent call last):
...
ValueError: a suitor with name "D" already exists
```

If we add a suitor without passing a name then the name of the suitor will not use one that is already chosen:

```
sage: suit = {0: (-1,  -2),
....:         2: (-2, -1)}
sage: revr = {-1: (0, 1),
....:         -2: (1, 0)}
sage: g = MatchingGame([suit, revr])
sage: g.suitors()
(0, 2)

sage: g.add_suitor()
sage: g.suitors()
(0, 2, 3)
```

bipartite_graph()

Construct a BipartiteGraph Object of the game. This method is similar to the plot method. Note that the game must be solved for this to work.

EXAMPLES:

An error is returned if the game is not solved:

```
sage: suit = {0: (3, 4),
....:         1: (3, 4)}
sage: revr = {3: (0, 1),
....:         4: (1, 0)}
sage: g = MatchingGame([suit, revr])
```

(continues on next page)

(continued from previous page)

```
sage: g.bipartite_graph()
Traceback (most recent call last):
...
ValueError: game has not been solved yet

sage: g.solve()
{0: 3, 1: 4}
sage: g.bipartite_graph()
Bipartite graph on 4 vertices
```

plot()

Create the plot representing the stable matching for the game. Note that the game must be solved for this to work.

EXAMPLES:

An error is returned if the game is not solved:

```
sage: suit = {0: (3, 4),
....:         1: (3, 4)}
sage: revr = {3: (0, 1),
....:         4: (1, 0)}
sage: g = MatchingGame([suit, revr])
sage: plot(g)
Traceback (most recent call last):
...
ValueError: game has not been solved yet

sage: g.solve()
{0: 3, 1: 4}
sage: plot(g)
Graphics object consisting of 7 graphics primitives
```

reviewers()

Return the reviewers of `self`.

EXAMPLES:

```
sage: g = MatchingGame(2)
sage: g.reviewers()
(-1, -2)
```

solve(*invert=False*)

Compute a stable matching for the game using the Gale-Shapley algorithm.

EXAMPLES:

```
sage: suitr_pref = {'J': ('A', 'D', 'C', 'B'),
....:               'K': ('A', 'B', 'C', 'D'),
....:               'L': ('B', 'C', 'D', 'A'),
....:               'M': ('C', 'A', 'B', 'D')}
sage: reviewr_pref = {'A': ('L', 'J', 'K', 'M'),
....:                 'B': ('J', 'M', 'L', 'K'),
....:                 'C': ('M', 'K', 'L', 'J'),
....:                 'D': ('M', 'K', 'J', 'L')}
sage: m = MatchingGame([suitr_pref, reviewr_pref])
sage: m.solve()
{'J': 'A', 'K': 'D', 'L': 'B', 'M': 'C'}
```

(continues on next page)

(continued from previous page)

```
sage: suitr_pref = {'J': ('A', 'D', 'C', 'B'),
....:               'K': ('A', 'B', 'C', 'D'),
....:               'L': ('B', 'C', 'D', 'A'),
....:               'M': ('C', 'A', 'B', 'D')}
sage: reviewr_pref = {'A': ('L', 'J', 'K', 'M'),
....:                 'B': ('J', 'M', 'L', 'K'),
....:                 'C': ('M', 'K', 'L', 'J'),
....:                 'D': ('M', 'K', 'J', 'L')}
sage: m = MatchingGame([suitr_pref, reviewr_pref])
sage: m.solve(invert=True)
{'A': 'L', 'B': 'J', 'C': 'M', 'D': 'K'}

sage: suitr_pref = {1: (-1,)}
sage: reviewr_pref = {-1: (1,)}
sage: m = MatchingGame([suitr_pref, reviewr_pref])
sage: m.solve()
{1: -1}

sage: suitr_pref = {}
sage: reviewr_pref = {}
sage: m = MatchingGame([suitr_pref, reviewr_pref])
sage: m.solve()
{}
```

suitors()

Return the suitors of `self`.

EXAMPLES:

```
sage: g = MatchingGame(2)
sage: g.suitors()
(1, 2)
```

class sage.game_theory.matching_game.**Player**(*name*)

Bases: `object`

A class to act as a data holder for the players used of the matching games.

These instances are used when initiating players and to keep track of whether or not partners have a preference.

CHAPTER

THREE

NORMAL FORM GAMES WITH N PLAYERS.

This module implements a class for normal form games (strategic form games) [**?**]. At present the following algorithms are implemented to compute equilibria of these games:

- `'enumeration'` - An implementation of the support enumeration algorithm built in Sage.
- `'LCP'` - An interface with the 'gambit' solver's implementation of the Lemke-Howson algorithm.
- `'lp'` - A built-in Sage implementation (with a gambit alternative) of a zero-sum game solver using linear programming. See `MixedIntegerLinearProgram` for more on MILP solvers in Sage.
- `'lrs'` - A solver interfacing with the 'lrslib' library.

The architecture for the class is based on the gambit architecture to ensure an easy transition between gambit and Sage. At present the algorithms for the computation of equilibria only solve 2 player games.

A very simple and well known example of normal form game is referred to as the 'Battle of the Sexes' in which two players Amy and Bob are modeled. Amy prefers to play video games and Bob prefers to watch a movie. They both however want to spend their evening together. This can be modeled using the following two matrices:

$$A = \begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix}$$
$$B = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$$

Matrix A represents the utilities of Amy and matrix B represents the utility of Bob. The choices of Amy correspond to the rows of the matrices:

- The first row corresponds to video games.
- The second row corresponds to movies.

Similarly Bob's choices are represented by the columns:

- The first column corresponds to video games.
- The second column corresponds to movies.

Thus, if both Amy and Bob choose to play video games: Amy receives a utility of 3 and Bob a utility of 2. If Amy is indeed going to stick with video games Bob has no incentive to deviate (and vice versa).

This situation repeats itself if both Amy and Bob choose to watch a movie: neither has an incentive to deviate.

This loosely described situation is referred to as a Nash Equilibrium. We can use Sage to find them, and more importantly, see if there is any other situation where Amy and Bob have no reason to change their choice of action:

Here is how we create the game in Sage:

```
sage: A = matrix([[3, 1], [0, 2]])
sage: B = matrix([[2, 1], [0, 3]])
sage: battle_of_the_sexes = NormalFormGame([A, B])
sage: battle_of_the_sexes
Normal Form Game with the following utilities: {(0, 0): [3, 2], (0, 1): [1, 1], (1,␣
↪0): [0, 0], (1, 1): [2, 3]}
```

To obtain the Nash equilibria we run the `obtain_nash()` method. In the first few examples, we will use the 'support enumeration' algorithm. A discussion about the different algorithms will be given later:

```
sage: battle_of_the_sexes.obtain_nash(algorithm='enumeration')
[[(0, 1), (0, 1)], [(3/4, 1/4), (1/4, 3/4)], [(1, 0), (1, 0)]]
```

If we look a bit closer at our output we see that a list of three pairs of tuples have been returned. Each of these correspond to a Nash Equilibrium, represented as a probability distribution over the available strategies:

- $[(1,0),(1,0)]$ corresponds to the first player only playing their first strategy and the second player also only playing their first strategy. In other words Amy and Bob both play video games.
- $[(0,1),(0,1)]$ corresponds to the first player only playing their second strategy and the second player also only playing their second strategy. In other words Amy and Bob both watch movies.
- $[(3/4,1/4),(1/4,3/4)]$ corresponds to players *mixing* their strategies. Amy plays video games 75% of the time and Bob watches movies 75% of the time. At this equilibrium point Amy and Bob will only ever do the same activity $3/8$ of the time.

We can use Sage to compute the expected utility for any mixed strategy pair (σ_1, σ_2). The payoff to player 1 is given by the vector/matrix multiplication:

$$\sigma_1 A \sigma_2$$

The payoff to player 2 is given by:

$$\sigma_1 B \sigma_2$$

To compute this in Sage we have:

```
sage: for ne in battle_of_the_sexes.obtain_nash(algorithm='enumeration'):
....:     print("Utility for {}: ".format(ne))
....:     print("{} {}".format(vector(ne[0]) * A * vector(ne[1]), vector(ne[0]) * B *␣
↪vector(ne[1])))
Utility for [(0, 1), (0, 1)]:
2 3
Utility for [(3/4, 1/4), (1/4, 3/4)]:
3/2 3/2
Utility for [(1, 0), (1, 0)]:
3 2
```

Allowing players to play mixed strategies ensures that there will always be a Nash Equilibrium for a normal form game. This result is called Nash's Theorem ([**?**]).

Let us consider the game called 'matching pennies' where two players each present a coin with either HEADS or TAILS showing. If the coins show the same side then player 1 wins, otherwise player 2 wins:

$$A = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$
$$B = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}$$

It should be relatively straightforward to observe, that there is no situation, where both players always do the same thing, and have no incentive to deviate.

We can plot the utility of player 1 when player 2 is playing a mixed strategy $\sigma_2 = (y, 1-y)$ (so that the utility to player 1 for playing strategy number i is given by the matrix/vector multiplication $(Ay)_i$, ie element in position i of the matrix/vector multiplication Ay)

```
sage: y = var('y')
sage: A = matrix([[1, -1], [-1, 1]])
sage: p = plot((A * vector([y, 1 - y]))[0], y, 0, 1, color='blue', legend_label='$u_
↪1(r_1, (y, 1-y))$', axes_labels=['$y$', ''])
sage: p += plot((A * vector([y, 1 - y]))[1], y, 0, 1, color='red', legend_label='$u_
↪1(r_2, (y, 1-y))$'); p
Graphics object consisting of 2 graphics primitives
```

We see that the only point at which player 1 is indifferent amongst the available strategies is when $y = 1/2$.

If we compute the Nash equilibria we see that this corresponds to a point at which both players are indifferent:

```
sage: A = matrix([[1, -1], [-1, 1]])
sage: B = matrix([[-1, 1], [1, -1]])
sage: matching_pennies = NormalFormGame([A, B])
sage: matching_pennies.obtain_nash(algorithm='enumeration')
[[(1/2, 1/2), (1/2, 1/2)]]
```

The utilities to both players at this Nash equilibrium is easily computed:

```
sage: [vector([1/2, 1/2]) * M * vector([1/2, 1/2])
....:  for M in matching_pennies.payoff_matrices()]
[0, 0]
```

Note that the above uses the `payoff_matrices` method which returns the payoff matrices for a 2 player game:

```
sage: matching_pennies.payoff_matrices()
(
[ 1 -1]  [-1  1]
[-1  1], [ 1 -1]
)
```

One can also input a single matrix and then a zero sum game is constructed. Here is an instance of Rock-Paper-Scissors-Lizard-Spock:

```
sage: A = matrix([[0, -1, 1, 1, -1],
....:             [1, 0, -1, -1, 1],
....:             [-1, 1, 0, 1 , -1],
....:             [-1, 1, -1, 0, 1],
....:             [1, -1, 1, -1, 0]])
sage: g = NormalFormGame([A])
sage: g.obtain_nash(algorithm='enumeration')
[[(1/5, 1/5, 1/5, 1/5, 1/5), (1/5, 1/5, 1/5, 1/5, 1/5)]]
```

We can also study games where players aim to minimize their utility. Here is the Prisoner's Dilemma (where players are aiming to reduce time spent in prison):

```
sage: A = matrix([[2, 5], [0, 4]])
sage: B = matrix([[2, 0], [5, 4]])
sage: prisoners_dilemma = NormalFormGame([A, B])
sage: prisoners_dilemma.obtain_nash(algorithm='enumeration', maximization=False)
[[(0, 1), (0, 1)]]
```

When obtaining Nash equilibrium the following algorithms are currently available:

- `'lp'`: A solver for constant sum 2 player games using linear programming. This constructs a `MixedIntegerLinearProgram` using the solver which was passed in with `solver` to solve the linear programming representation of the game. See `MixedIntegerLinearProgram` for more on MILP solvers in Sage.
- `'lrs'`: Reverse search vertex enumeration for 2 player games. This algorithm uses the optional 'lrslib' package. To install it, type `sage -i lrslib` in the shell. For more information, see [**?**].
- `'LCP'`: Linear complementarity program algorithm for 2 player games. This algorithm uses the open source game theory package: Gambit [**?**]. At present this is the only gambit algorithm available in sage but further development will hope to implement more algorithms (in particular for games with more than 2 players). To install it, type `sage -i gambit` in the shell.
- `'enumeration'`: Support enumeration for 2 player games. This algorithm is hard coded in Sage and checks through all potential supports of a strategy. Supports of a given size with a conditionally dominated strategy are ignored. Note: this is not the preferred algorithm. The algorithm implemented is a combination of a basic algorithm described in [**?**] and a pruning component described in [**?**].

Below we show how the these algorithms are called:

```
sage: matching_pennies.obtain_nash(algorithm='lrs')  # optional - lrslib
[[(1/2, 1/2), (1/2, 1/2)]]
sage: matching_pennies.obtain_nash(algorithm='LCP')  # optional - gambit
[[(0.5, 0.5), (0.5, 0.5)]]
sage: matching_pennies.obtain_nash(algorithm='lp', solver='PPL')
[[(1/2, 1/2), (1/2, 1/2)]]
sage: matching_pennies.obtain_nash(algorithm='lp', solver='gambit') # optional -␣
↪gambit
[[(0.5, 0.5), (0.5, 0.5)]]
sage: matching_pennies.obtain_nash(algorithm='enumeration')
[[(1/2, 1/2), (1/2, 1/2)]]
```

Note that if no algorithm argument is passed then the default will be selected according to the following order (if the corresponding package is installed):

1. `'lp'` (if the game is constant-sum; uses the solver chosen by Sage)
2. `'lrs'` (requires 'lrslib')
3. `'enumeration'`

Here is a game being constructed using gambit syntax (note that a `NormalFormGame` object acts like a dictionary with pure strategy tuples as keys and payoffs as their values):

```
sage: f = NormalFormGame()
sage: f.add_player(2)  # Adding first player with 2 strategies
sage: f.add_player(2)  # Adding second player with 2 strategies
sage: f[0,0][0] = 1
sage: f[0,0][1] = 3
sage: f[0,1][0] = 2
sage: f[0,1][1] = 3
sage: f[1,0][0] = 3
sage: f[1,0][1] = 1
sage: f[1,1][0] = 4
sage: f[1,1][1] = 4
sage: f
Normal Form Game with the following utilities: {(0, 0): [1, 3], (0, 1): [2, 3], (1,␣
↪0): [3, 1], (1, 1): [4, 4]}
```

Once this game is constructed we can view the payoff matrices and solve the game:

```
sage: f.payoff_matrices()
(
[1 2]  [3 3]
[3 4], [1 4]
)
sage: f.obtain_nash(algorithm='enumeration')
[[(0, 1), (0, 1)]]
```

We can add an extra strategy to the first player:

```
sage: f.add_strategy(0)
sage: f
Normal Form Game with the following utilities: {(0, 0): [1, 3],
 (0, 1): [2, 3],
 (1, 0): [3, 1],
 (1, 1): [4, 4],
 (2, 0): [False, False],
 (2, 1): [False, False]}
```

If we do this and try and obtain the Nash equilibrium or view the payoff matrices(without specifying the utilities), an error is returned:

```
sage: f.obtain_nash()
Traceback (most recent call last):
...
ValueError: utilities have not been populated
sage: f.payoff_matrices()
Traceback (most recent call last):
...
ValueError: utilities have not been populated
```

Here we populate the missing utilities:

```
sage: f[2, 1] = [5, 3]
sage: f[2, 0] = [2, 1]
sage: f.payoff_matrices()
(
[1 2]  [3 3]
[3 4]  [1 4]
[2 5], [1 3]
)
sage: f.obtain_nash()
[[(0, 0, 1), (0, 1)]]
```

We can use the same syntax as above to create games with more than 2 players:

```
sage: threegame = NormalFormGame()
sage: threegame.add_player(2)  # Adding first player with 2 strategies
sage: threegame.add_player(2)  # Adding second player with 2 strategies
sage: threegame.add_player(2)  # Adding third player with 2 strategies
sage: threegame[0, 0, 0][0] = 3
sage: threegame[0, 0, 0][1] = 1
sage: threegame[0, 0, 0][2] = 4
sage: threegame[0, 0, 1][0] = 1
sage: threegame[0, 0, 1][1] = 5
sage: threegame[0, 0, 1][2] = 9
```

(continues on next page)

(continued from previous page)

```
sage: threegame[0, 1, 0][0] = 2
sage: threegame[0, 1, 0][1] = 6
sage: threegame[0, 1, 0][2] = 5
sage: threegame[0, 1, 1][0] = 3
sage: threegame[0, 1, 1][1] = 5
sage: threegame[0, 1, 1][2] = 8
sage: threegame[1, 0, 0][0] = 9
sage: threegame[1, 0, 0][1] = 7
sage: threegame[1, 0, 0][2] = 9
sage: threegame[1, 0, 1][0] = 3
sage: threegame[1, 0, 1][1] = 2
sage: threegame[1, 0, 1][2] = 3
sage: threegame[1, 1, 0][0] = 8
sage: threegame[1, 1, 0][1] = 4
sage: threegame[1, 1, 0][2] = 6
sage: threegame[1, 1, 1][0] = 2
sage: threegame[1, 1, 1][1] = 6
sage: threegame[1, 1, 1][2] = 4
sage: threegame
Normal Form Game with the following utilities: {(0, 0, 0): [3, 1, 4],
 (0, 0, 1): [1, 5, 9],
 (0, 1, 0): [2, 6, 5],
 (0, 1, 1): [3, 5, 8],
 (1, 0, 0): [9, 7, 9],
 (1, 0, 1): [3, 2, 3],
 (1, 1, 0): [8, 4, 6],
 (1, 1, 1): [2, 6, 4]}
```

The above requires a lot of input that could be simplified if there is another data structure with our utilities and/or a structure to the utilities. The following example creates a game with a relatively strange utility function:

```
sage: def utility(strategy_triplet, player):
....:     return sum(strategy_triplet) * player
sage: threegame = NormalFormGame()
sage: threegame.add_player(2)  # Adding first player with 2 strategies
sage: threegame.add_player(2)  # Adding second player with 2 strategies
sage: threegame.add_player(2)  # Adding third player with 2 strategies
sage: for i, j, k in [(i, j, k) for i in [0,1] for j in [0,1] for k in [0,1]]:
....:     for p in range(3):
....:         threegame[i, j, k][p] = utility([i, j, k], p)
sage: threegame
Normal Form Game with the following utilities: {(0, 0, 0): [0, 0, 0],
 (0, 0, 1): [0, 1, 2],
 (0, 1, 0): [0, 1, 2],
 (0, 1, 1): [0, 2, 4],
 (1, 0, 0): [0, 1, 2],
 (1, 0, 1): [0, 2, 4],
 (1, 1, 0): [0, 2, 4],
 (1, 1, 1): [0, 3, 6]}
```

At present no algorithm has been implemented in Sage for games with more than 2 players:

```
sage: threegame.obtain_nash()
Traceback (most recent call last):
...
NotImplementedError: Nash equilibrium for games with more than 2 players have not␣
↪been implemented yet. Please see the gambit website (http://gambit.sourceforge.net/
↪) that has a variety of available algorithms
```

(continues on next page)

(continued from previous page)

There are however a variety of such algorithms available in gambit, further compatibility between Sage and gambit is actively being developed: https://github.com/tturocy/gambit/tree/sage_integration.

It can be shown that linear scaling of the payoff matrices conserves the equilibrium values:

```
sage: A = matrix([[2, 1], [1, 2.5]])
sage: B = matrix([[-1, 3], [2, 1]])
sage: g = NormalFormGame([A, B])
sage: g.obtain_nash(algorithm='enumeration')
[[(1/5, 4/5), (3/5, 2/5)]]
sage: g.obtain_nash(algorithm='lrs') # optional - lrslib
[[(1/5, 4/5), (3/5, 2/5)]]
sage: A = 2 * A
sage: g = NormalFormGame([A, B])
sage: g.obtain_nash(algorithm='LCP')  # optional - gambit
[[(0.2, 0.8), (0.6, 0.4)]]
```

It is also possible to generate a Normal form game from a gambit Game:

```
sage: from gambit import Game  # optional - gambit
sage: gambitgame= Game.new_table([2, 2])  # optional - gambit
sage: gambitgame[int(0), int(0)][int(0)] = int(8)  # optional - gambit
sage: gambitgame[int(0), int(0)][int(1)] = int(8)  # optional - gambit
sage: gambitgame[int(0), int(1)][int(0)] = int(2)  # optional - gambit
sage: gambitgame[int(0), int(1)][int(1)] = int(10)  # optional - gambit
sage: gambitgame[int(1), int(0)][int(0)] = int(10)  # optional - gambit
sage: gambitgame[int(1), int(0)][int(1)] = int(2)  # optional - gambit
sage: gambitgame[int(1), int(1)][int(0)] = int(5)  # optional - gambit
sage: gambitgame[int(1), int(1)][int(1)] = int(5)  # optional - gambit
sage: g = NormalFormGame(gambitgame)  # optional - gambit
sage: g  # optional - gambit
Normal Form Game with the following utilities: {(0, 0): [8.0, 8.0],
 (0, 1): [2.0, 10.0],
 (1, 0): [10.0, 2.0],
 (1, 1): [5.0, 5.0]}
```

For more information on using Gambit in Sage see: *Using Gambit in Sage*. This includes how to access Gambit directly using the version of iPython shipped with Sage and an explanation as to why the `int` calls are needed to handle the Sage preparser.

Here is a slightly longer game that would take too long to solve with `'enumeration'`. Consider the following:

An airline loses two suitcases belonging to two different travelers. Both suitcases happen to be identical and contain identical antiques. An airline manager tasked to settle the claims of both travelers explains that the airline is liable for a maximum of 10 per suitcase, and in order to determine an honest appraised value of the antiques the manager separates both travelers so they can't confer, and asks them to write down the amount of their value at no less than 2 and no larger than 10. He also tells them that if both write down the same number, he will treat that number as the true dollar value of both suitcases and reimburse both travelers that amount.

However, if one writes down a smaller number than the other, this smaller number will be taken as the true dollar value, and both travelers will receive that amount along with a bonus/malus: 2 extra will be paid to the traveler who wrote down the lower value and a 2 deduction will be taken from the person who wrote down the higher amount. The challenge is: what strategy should both travelers follow to decide the value they should write down?

In the following we create the game (with a max value of 10) and solve it:

```
sage: K = 10  # Modifying this value lets us play with games of any size
sage: A = matrix([[min(i,j) + 2 * sign(j-i)  for j in range(K, 1, -1)]
....:             for i in range(K, 1, -1)])
sage: B = matrix([[min(i,j) + 2 * sign(i-j)  for j in range(K, 1, -1)]
....:             for i in range(K, 1, -1)])
sage: g = NormalFormGame([A, B])
sage: g.obtain_nash(algorithm='lrs') # optional - lrslib
[[(0, 0, 0, 0, 0, 0, 0, 0, 1), (0, 0, 0, 0, 0, 0, 0, 0, 1)]]
sage: g.obtain_nash(algorithm='LCP') # optional - gambit
[[(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0),
  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)]]
```

The output is a pair of vectors (as before) showing the Nash equilibrium. In particular it here shows that out of the 10 possible strategies both players should choose the last. Recall that the above considers a reduced version of the game where individuals can claim integer values from 10 to 2. The equilibrium strategy is thus for both players to state that the value of their suitcase is 2.

Several standard Normal Form Games have also been implemented. For more information on how to access these, see: `Game Theory Catalog`. Included is information on the situation each Game models. For example:

```
sage: g = game_theory.normal_form_games.PrisonersDilemma()
sage: g
Prisoners dilemma - Normal Form Game with the following utilities: ...
sage: d = {(0, 1): [-5, 0], (1, 0): [0, -5],
....:      (0, 0): [-2, -2], (1, 1): [-4, -4]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 1), (0, 1)]]
```

We can easily obtain the best response for a player to a given strategy. In this example we obtain the best responses for Player 1, when Player 2 uses two different strategies:

```
sage: A = matrix([[3, 0], [0, 3], [1.5, 1.5]])
sage: B = matrix([[4, 3], [2, 6], [3, 1]])
sage: g = NormalFormGame([A, B])
sage: g.best_responses((1/2, 1/2), player=0)
[0, 1, 2]
sage: g.best_responses((3/4, 1/4), player=0)
[0]
```

Here we do the same for player 2:

```
sage: g.best_responses((4/5, 1/5, 0), player=1)
[0, 1]
```

We see that for the game Rock-Paper-Scissors-Lizard-Spock any pure strategy has two best responses:

```
sage: g = game_theory.normal_form_games.RPSLS()
sage: A, B = g.payoff_matrices()
sage: A, B
(
[ 0 -1  1  1 -1]  [ 0  1 -1 -1  1]
[ 1  0 -1 -1  1]  [-1  0  1  1 -1]
[-1  1  0  1 -1]  [ 1 -1  0 -1  1]
[-1  1 -1  0  1]  [ 1 -1  1  0 -1]
[ 1 -1  1 -1  0], [-1  1 -1  1  0]
```

(continues on next page)

(continued from previous page)

```
)
sage: g.best_responses((1, 0, 0, 0, 0), player=0)
[1, 4]
sage: g.best_responses((0, 1, 0, 0, 0), player=0)
[2, 3]
sage: g.best_responses((0, 0, 1, 0, 0), player=0)
[0, 4]
sage: g.best_responses((0, 0, 0, 1, 0), player=0)
[0, 2]
sage: g.best_responses((0, 0, 0, 0, 1), player=0)
[1, 3]
sage: g.best_responses((1, 0, 0, 0, 0), player=1)
[1, 4]
sage: g.best_responses((0, 1, 0, 0, 0), player=1)
[2, 3]
sage: g.best_responses((0, 0, 1, 0, 0), player=1)
[0, 4]
sage: g.best_responses((0, 0, 0, 1, 0), player=1)
[0, 2]
sage: g.best_responses((0, 0, 0, 0, 1), player=1)
[1, 3]
```

Note that degenerate games can cause problems for most algorithms. The following example in fact has an infinite quantity of equilibria which is evidenced by the various algorithms returning different solutions:

```
sage: A = matrix([[3,3],[2,5],[0,6]])
sage: B = matrix([[3,3],[2,6],[3,1]])
sage: degenerate_game = NormalFormGame([A,B])
sage: degenerate_game.obtain_nash(algorithm='lrs') # optional - lrslib
[[(0, 1/3, 2/3), (1/3, 2/3)], [(1, 0, 0), (1/2, 3)], [(1, 0, 0), (1, 3)]]
sage: degenerate_game.obtain_nash(algorithm='LCP') # optional - gambit
[[(0.0, 0.3333333333, 0.6666666667), (0.3333333333, 0.6666666667)],
 [(1.0, -0.0, 0.0), (0.6666666667, 0.3333333333)],
 [(1.0, 0.0, 0.0), (1.0, 0.0)]]
sage: degenerate_game.obtain_nash(algorithm='enumeration')
[[(0, 1/3, 2/3), (1/3, 2/3)], [(1, 0, 0), (1, 0)]]
```

We can check the cause of this by using `is_degenerate()`:

```
sage: degenerate_game.is_degenerate()
True
```

Note the 'negative' -0.0 output by gambit. This is due to the numerical nature of the algorithm used.

Here is an example with the trivial game where all payoffs are 0:

```
sage: g = NormalFormGame()
sage: g.add_player(3)  # Adding first player with 3 strategies
sage: g.add_player(3)  # Adding second player with 3 strategies
sage: for key in g:
....:     g[key] = [0, 0]
sage: g.payoff_matrices()
(
[0 0 0]  [0 0 0]
[0 0 0]  [0 0 0]
[0 0 0], [0 0 0]
)
```

(continues on next page)

(continued from previous page)

```
sage: g.obtain_nash(algorithm='enumeration')
[[(0, 0, 1), (0, 0, 1)], [(0, 0, 1), (0, 1, 0)], [(0, 0, 1), (1, 0, 0)],
 [(0, 1, 0), (0, 0, 1)], [(0, 1, 0), (0, 1, 0)], [(0, 1, 0), (1, 0, 0)],
 [(1, 0, 0), (0, 0, 1)], [(1, 0, 0), (0, 1, 0)], [(1, 0, 0), (1, 0, 0)]]
```

A good description of degenerate games can be found in [**?**].

REFERENCES:

- [**?**]
- [**?**]
- [**?**]
- [**?**]
- [**?**]

AUTHOR:

- James Campbell and Vince Knight (06-2014): Original version
- Tobenna P. Igwe: Constant-sum game solvers

class sage.game_theory.normal_form_game.**NormalFormGame**(*generator=None*)

Bases: sage.structure.sage_object.SageObject, _abcoll.MutableMapping

An object representing a Normal Form Game. Primarily used to compute the Nash Equilibria.

INPUT:

- `generator` – can be a list of 2 matrices, a single matrix or left blank

add_player(*num_strategies*)

Add a player to a NormalFormGame.

INPUT:

- `num_strategies` – the number of strategies the player should have

EXAMPLES:

```
sage: g = NormalFormGame()
sage: g.add_player(2)  # Adding first player with 2 strategies
sage: g.add_player(1)  # Adding second player with 1 strategy
sage: g.add_player(1)  # Adding third player with 1 strategy
sage: g
Normal Form Game with the following utilities: {(0, 0, 0): [False, False,␣
↪False], (1, 0, 0): [False, False, False]}
```

add_strategy(*player*)

Add a strategy to a player, will not affect already completed strategy profiles.

INPUT:

- `player` – the index of the player

EXAMPLES:

A simple example:

```
sage: s = matrix([[1, 0], [-2, 3]])
sage: t = matrix([[3, 2], [-1, 0]])
sage: example = NormalFormGame([s, t])
sage: example
Normal Form Game with the following utilities: {(0, 0): [1, 3], (0, 1): [0,␣
↪2], (1, 0): [-2, -1], (1, 1): [3, 0]}
sage: example.add_strategy(0)
sage: example
Normal Form Game with the following utilities: {(0, 0): [1, 3],
 (0, 1): [0, 2],
 (1, 0): [-2, -1],
 (1, 1): [3, 0],
 (2, 0): [False, False],
 (2, 1): [False, False]}
```

best_responses(*strategy*, *player*)

For a given strategy for a player and the index of the opponent, computes the payoff for the opponent and returns a list of the indices of the best responses. Only implemented for two player games

INPUT:

- `strategy` – a probability distribution vector
- `player` – the index of the opponent, `0` for the row player, `1` for the column player.

EXAMPLES:

```
sage: A = matrix([[3, 0], [0, 3], [1.5, 1.5]])
sage: B = matrix([[4, 3], [2, 6], [3, 1]])
sage: g = NormalFormGame([A, B])
```

Now we can obtain the best responses for Player 1, when Player 2 uses different strategies:

```
sage: g.best_responses((1/2, 1/2), player=0)
[0, 1, 2]
sage: g.best_responses((3/4, 1/4), player=0)
[0]
```

To get the best responses for Player 2 we pass the argument `player=1`

sage: g.best_responses((4/5, 1/5, 0), player=1) [0, 1]

sage: A = matrix([[1, 0], [0, 1], [0, 0]]) sage: B = matrix([[1, 0], [0, 1], [0.7, 0.8]]) sage: g = NormalFormGame([A, B]) sage: g.best_responses((0, 1, 0), player=1) [1]

sage: A = matrix([[3,3],[2,5],[0,6]]) sage: B = matrix([[3,3],[2,6],[3,1]]) sage: degenerate_game = NormalFormGame([A,B]) sage: degenerate_game.best_responses((1, 0, 0), player=1) [0, 1]

sage: A = matrix([[3, 0], [0, 3], [1.5, 1.5]]) sage: B = matrix([[4, 3], [2, 6], [3, 1]]) sage: g = NormalFormGame([A, B]) sage: g.best_responses((1/3, 1/3, 1/3), player=1) [1]

Note that this has only been implemented for 2 player games:

```
sage: g = NormalFormGame()
sage: g.add_player(2)  # adding first player with 2 strategies
sage: g.add_player(2)  # adding second player with 2 strategies
sage: g.add_player(2)  # adding third player with 2 strategies
sage: g.best_responses((1/2, 1/2), player=2)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: Only available for 2 player games
```

If the strategy is not of the correct dimension for the given player then an error is returned:

```
sage: A = matrix([[3, 0], [0, 3], [1.5, 1.5]])
sage: B = matrix([[4, 3], [2, 6], [3, 1]])
sage: g = NormalFormGame([A, B])
sage: g.best_responses((1/2, 1/2), player=1)
Traceback (most recent call last):
...
ValueError: Strategy is not of correct dimension

sage: g.best_responses((1/3, 1/3, 1/3), player=0)
Traceback (most recent call last):
...
ValueError: Strategy is not of correct dimension
```

If the strategy is not a true probability vector then an error is passed:

sage: A = matrix([[3, 0], [0, 3], [1.5, 1.5]]) sage: B = matrix([[4, 3], [2, 6], [3, 1]]) sage: g = NormalFormGame([A, B]) sage: g.best_responses((1/3, 1/2, 0), player=1) Traceback (most recent call last): ... ValueError: Strategy is not a probability distribution vector

sage: A = matrix([[3, 0], [0, 3], [1.5, 1.5]]) sage: B = matrix([[4, 3], [2, 6], [3, 1]]) sage: g = NormalFormGame([A, B]) sage: g.best_responses((3/2, -1/2), player=0) Traceback (most recent call last): ... ValueError: Strategy is not a probability distribution vector

If the player specified is not 0 or 1, an error is raised:

```
sage: A = matrix([[3, 0], [0, 3], [1.5, 1.5]])
sage: B = matrix([[4, 3], [2, 6], [3, 1]])
sage: g = NormalFormGame([A, B])
sage: g.best_responses((1/2, 1/2), player='Player1')
Traceback (most recent call last):
...
ValueError: Player1 is not an index of the opponent, must be 0 or 1
```

is_constant_sum()

Checks if the game is constant sum.

EXAMPLES:

```
sage: A = matrix([[2, 1], [1, 2.5]])
sage: g = NormalFormGame([A])
sage: g.is_constant_sum()
True
sage: g = NormalFormGame([A, A])
sage: g.is_constant_sum()
False
sage: A = matrix([[1, 1], [1, 1]])
sage: g = NormalFormGame([A, A])
sage: g.is_constant_sum()
True
sage: A = matrix([[1, 1, 2], [1, 1, -1], [1, -1, 1]])
sage: B = matrix([[2, 2, 1], [2, 2, 4], [2, 4, 2]])
sage: g = NormalFormGame([A, B])
```

(continues on next page)

(continued from previous page)

```
sage: g.is_constant_sum()
True
sage: A = matrix([[1, 1, 2], [1, 1, -1], [1, -1, 1]])
sage: B = matrix([[2, 2, 1], [2, 2.1, 4], [2, 4, 2]])
sage: g = NormalFormGame([A, B])
sage: g.is_constant_sum()
False
```

is_degenerate(*certificate=False*)

A function to check whether the game is degenerate or not. Will return a boolean.

A two-player game is called nondegenerate if no mixed strategy of support size k has more than k pure best responses [**?**]. In a degenerate game, this definition is violated, for example if there is a pure strategy that has two pure best responses.

The implementation here transforms the search over mixed strategies to a search over supports which is a discrete search. A full explanation of this is given in [**?**]. This problem is known to be NP-Hard [**?**]. Another possible implementation is via best response polytopes, see trac ticket #18958.

The game Rock-Paper-Scissors is an example of a non-degenerate game,:

```
sage: g = game_theory.normal_form_games.RPS()
sage: g.is_degenerate()
False
```

whereas Rock-Paper-Scissors-Lizard-Spock is degenerate because for every pure strategy there are two best responses.:

```
sage: g = game_theory.normal_form_games.RPSLS()
sage: g.is_degenerate()
True
```

EXAMPLES:

Here is an example of a degenerate game given in [**?**]:

```
sage: A = matrix([[3, 3], [2, 5], [0, 6]])
sage: B = matrix([[3, 3], [2, 6], [3, 1]])
sage: degenerate_game = NormalFormGame([A,B])
sage: degenerate_game.is_degenerate()
True
```

Here is an example of a degenerate game given in [**?**]:

```
sage: A = matrix([[0, 6], [2, 5], [3, 3]])
sage: B = matrix([[1, 0], [0, 2], [4, 4]])
sage: d_game = NormalFormGame([A, B])
sage: d_game.is_degenerate()
True
```

Here are some other examples of degenerate games:

```
sage: M = matrix([[2, 1], [1, 1]])
sage: N = matrix([[1, 1], [1, 2]])
sage: game  = NormalFormGame([M, N])
sage: game.is_degenerate()
True
```

If more information is required, it may be useful to use `certificate=True`. This will return a boolean of whether the game is degenerate or not, and if True; a tuple containing the strategy where degeneracy was found and the player it belongs to. `0` is the row player and `1` is the column player.:

```
sage: M = matrix([[2, 1], [1, 1]])
sage: N = matrix([[1, 1], [1, 2]])
sage: g  = NormalFormGame([M, N])
sage: test, certificate = g.is_degenerate(certificate=True)
sage: test, certificate
(True, ((1, 0), 0))
```

Using the output, we see that the opponent has more best responses than the size of the support of the strategy in question `(1, 0)`. (We specify the player as `(player + 1) % 2` to ensure that we have the opponent's index.):

```
sage: g.best_responses(certificate[0], (certificate[1] + 1) % 2)
[0, 1]
```

Another example with a mixed strategy causing degeneracy.:

```
sage: A = matrix([[3, 0], [0, 3], [1.5, 1.5]])
sage: B = matrix([[4, 3], [2, 6], [3, 1]])
sage: g = NormalFormGame([A, B])
sage: test, certificate = g.is_degenerate(certificate=True)
sage: test, certificate
(True, ((1/2, 1/2), 1))
```

Again, we see that the opponent has more best responses than the size of the support of the strategy in question `(1/2, 1/2)`.:

```
sage: g.best_responses(certificate[0], (certificate[1] + 1) % 2)
[0, 1, 2]
```

Sometimes, the different algorithms for obtaining nash_equilibria don't agree with each other. This can happen when games are degenerate:

```
sage: a = matrix([[-75, 18, 45, 33],
....:             [42, -8, -77, -18],
....:             [83, 18, 11, 40],
....:             [-10, -38, 76, -9]])
sage: b = matrix([[62, 64, 87, 51],
....:             [-41, -27, -69, 52],
....:             [-17, 25, -97, -82],
....:             [30, 31, -1, 50]])
sage: d_game = NormalFormGame([a, b])
sage: d_game.obtain_nash(algorithm='lrs') # optional - lrslib
[[(0, 0, 1, 0), (0, 1, 0, 0)],
 [(17/29, 0, 0, 12/29), (0, 0, 42/73, 31/73)],
 [(122/145, 0, 23/145, 0), (0, 1, 0, 0)]]
sage: d_game.obtain_nash(algorithm='LCP') # optional - gambit
[[(0.5862068966, 0.0, 0.0, 0.4137931034),
  (0.0, 0.0, 0.5753424658, 0.4246575342)]]
sage: d_game.obtain_nash(algorithm='enumeration')
[[(0, 0, 1, 0), (0, 1, 0, 0)], [(17/29, 0, 0, 12/29), (0, 0, 42/73, 31/73)]]
sage: d_game.is_degenerate()
True
```

obtain_nash(*algorithm=False*, *maximization=True*, *solver=None*)

A function to return the Nash equilibrium for the game. Optional arguments can be used to specify the algorithm used. If no algorithm is passed then an attempt is made to use the most appropriate algorithm.

INPUT:

- `algorithm` - the following algorithms should be available through this function:
  - `'lrs'` - This algorithm is only suited for 2 player games. See the lrs web site (http://cgm.cs.mcgill.ca/~avis/C/lrs.html).
  - `'LCP'` - This algorithm is only suited for 2 player games. See the gambit web site (http://gambit.sourceforge.net/).
  - `'lp'` - This algorithm is only suited for 2 player constant sum games. Uses MILP solver determined by the `solver` argument.
  - `'enumeration'` - This is a very inefficient algorithm (in essence a brute force approach).
    1. For each k in 1... min(size of strategy sets)
    2. For each I,J supports of size k
    3. Prune: check if supports are dominated
    4. Solve indifference conditions and check that have Nash Equilibrium.

    Solving the indifference conditions is done by building the corresponding linear system. If ρ_1, ρ_2 are the supports player 1 and 2 respectively. Then, indifference implies:

    $$u_1(s_1, \rho_2) = u_1(s_2, \rho_2)$$

    for all s_1, s_2 in the support of ρ_1. This corresponds to:

    $$\sum_{j \in S(\rho_2)} A_{s_1,j}\rho_{2j} = \sum_{j \in S(\rho_2)} A_{s_2,j}\rho_{2j}$$

    for all s_1, s_2 in the support of ρ_1 where A is the payoff matrix of player 1. Equivalently we can consider consecutive rows of A (instead of all pairs of strategies). Thus the corresponding linear system can be written as:

    $$\left(\sum_{j \in S(\rho_2)} A_{i,j} - A_{i+1,j}\right)\rho_{2j}$$

    for all $1 \leq i \leq |S(\rho_1)|$ (where A has been modified to only contain the rows corresponding to $S(\rho_1)$). We also require all elements of ρ_2 to sum to 1:

    $$\sum_{j \in S(\rho_1)} \rho_{2j} = 1$$

- `maximization` – (default: `True`) whether a player is trying to maximize their utility or minimize it:
  - When set to `True` it is assumed that players aim to maximise their utility.
  - When set to `False` it is assumed that players aim to minimise their utility.
- `solver` – (optional) see `MixedIntegerLinearProgram` for more information on the MILP solvers in Sage, may also be `'gambit'` to use the MILP solver included with the gambit library. Note that `None` means to use the default Sage LP solver, normally GLPK.

EXAMPLES:

A game with 1 equilibrium when `maximization` is `True` and 3 when `maximization` is `False`:

```
sage: A = matrix([[10, 500, 44],
....:       [15, 10, 105],
....:       [19, 204, 55],
....:       [20, 200, 590]])
sage: B = matrix([[2, 1, 2],
....:             [0, 5, 6],
....:             [3, 4, 1],
....:             [4, 1, 20]])
sage: g=NormalFormGame([A, B])
sage: g.obtain_nash(algorithm='lrs') # optional - lrslib
[[(0, 0, 0, 1), (0, 0, 1)]]
sage: g.obtain_nash(algorithm='lrs', maximization=False) # optional - lrslib
[[(2/3, 1/12, 1/4, 0), (6333/8045, 247/8045, 293/1609)], [(3/4, 0, 1/4, 0),␣
↪(0, 11/307, 296/307)], [(5/6, 1/6, 0, 0), (98/99, 1/99, 0)]]
```

This particular game has 3 Nash equilibria:

```
sage: A = matrix([[3,3],
....:             [2,5],
....:             [0,6]])
sage: B = matrix([[3,2],
....:             [2,6],
....:             [3,1]])
sage: g = NormalFormGame([A, B])
sage: g.obtain_nash(algorithm='enumeration')
[[(0, 1/3, 2/3), (1/3, 2/3)], [(4/5, 1/5, 0), (2/3, 1/3)], [(1, 0, 0), (1,␣
↪0)]]
```

Here is a slightly larger game:

```
sage: A = matrix([[160, 205, 44],
....:             [175, 180, 45],
....:             [201, 204, 50],
....:             [120, 207, 49]])
sage: B = matrix([[2, 2, 2],
....:             [1, 0, 0],
....:             [3, 4, 1],
....:             [4, 1, 2]])
sage: g=NormalFormGame([A, B])
sage: g.obtain_nash(algorithm='enumeration')
[[(0, 0, 3/4, 1/4), (1/28, 27/28, 0)]]
sage: g.obtain_nash(algorithm='lrs')  # optional - lrslib
[[(0, 0, 3/4, 1/4), (1/28, 27/28, 0)]]
sage: g.obtain_nash(algorithm='LCP')  # optional - gambit
[[(0.0, 0.0, 0.75, 0.25), (0.0357142857, 0.9642857143, 0.0)]]
```

2 random matrices:

```
sage: player1 = matrix([[2, 8, -1, 1, 0],
....:                   [1, 1, 2, 1, 80],
....:                   [0, 2, 15, 0, -12],
....:                   [-2, -2, 1, -20, -1],
....:                   [1, -2, -1, -2, 1]])
sage: player2 = matrix([[0, 8, 4, 2, -1],
....:                   [6, 14, -5, 1, 0],
....:                   [0, -2, -1, 8, -1],
....:                   [1, -1, 3, -3, 2],
```

(continues on next page)

(continued from previous page)

```
....:                   [8, -4, 1, 1, -17]])
sage: fivegame = NormalFormGame([player1, player2])
sage: fivegame.obtain_nash(algorithm='enumeration')
[[(1, 0, 0, 0, 0), (0, 1, 0, 0, 0)]]
sage: fivegame.obtain_nash(algorithm='lrs') # optional - lrslib
[[(1, 0, 0, 0, 0), (0, 1, 0, 0, 0)]]
sage: fivegame.obtain_nash(algorithm='LCP') # optional - gambit
[[(1.0, 0.0, 0.0, 0.0, 0.0), (0.0, 1.0, 0.0, 0.0, 0.0)]]
```

Here are some examples of finding Nash equilibria for constant-sum games:

```
sage: A = matrix.identity(2)
sage: cg = NormalFormGame([A])
sage: cg.obtain_nash(algorithm='lp')
[[(0.5, 0.5), (0.5, 0.5)]]
sage: cg.obtain_nash(algorithm='lp', solver='Coin') # optional - cbc
[[(0.5, 0.5), (0.5, 0.5)]]
sage: cg.obtain_nash(algorithm='lp', solver='PPL')
[[(1/2, 1/2), (1/2, 1/2)]]
sage: cg.obtain_nash(algorithm='lp', solver='gambit') # optional - gambit
[[(0.5, 0.5), (0.5, 0.5)]]
sage: A = matrix([[2, 1], [1, 3]])
sage: cg = NormalFormGame([A])
sage: ne = cg.obtain_nash(algorithm='lp', solver='glpk')
sage: [[[round(el, 6) for el in v] for v in eq] for eq in ne]
[[[0.666667, 0.333333], [0.666667, 0.333333]]]
sage: ne = cg.obtain_nash(algorithm='lp', solver='Coin') # optional - cbc
sage: [[[round(el, 6) for el in v] for v in eq] for eq in ne] # optional - cbc
[[[0.666667, 0.333333], [0.666667, 0.333333]]]
sage: cg.obtain_nash(algorithm='lp', solver='PPL')
[[(2/3, 1/3), (2/3, 1/3)]]
sage: ne = cg.obtain_nash(algorithm='lp', solver='gambit') # optional - gambit
sage: [[[round(el, 6) for el in v] for v in eq] for eq in ne] # optional -␣
↪gambit
[[[0.666667, 0.333333], [0.666667, 0.333333]]]
sage: A = matrix([[1, 2, 1], [1, 1, 2], [2, 1, 1]])
sage: B = matrix([[2, 1, 2], [2, 2, 1], [1, 2, 2]])
sage: cg = NormalFormGame([A, B])
sage: ne = cg.obtain_nash(algorithm='lp', solver='glpk')
sage: [[[round(el, 6) for el in v] for v in eq] for eq in ne]
[[[0.333333, 0.333333, 0.333333], [0.333333, 0.333333, 0.333333]]]
sage: ne = cg.obtain_nash(algorithm='lp', solver='Coin') # optional - cbc
sage: [[[round(el, 6) for el in v] for v in eq] for eq in ne] # optional - cbc
[[[0.333333, 0.333333, 0.333333], [0.333333, 0.333333, 0.333333]]]
sage: cg.obtain_nash(algorithm='lp', solver='PPL')
[[(1/3, 1/3, 1/3), (1/3, 1/3, 1/3)]]
sage: ne = cg.obtain_nash(algorithm='lp', solver='gambit') # optional - gambit
sage: [[[round(el, 6) for el in v] for v in eq] for eq in ne] # optional -␣
↪gambit
[[[0.333333, 0.333333, 0.333333], [0.333333, 0.333333, 0.333333]]]
sage: A = matrix([[160, 205, 44],
....:             [175, 180, 45],
....:             [201, 204, 50],
....:             [120, 207, 49]])
sage: cg = NormalFormGame([A])
sage: cg.obtain_nash(algorithm='lp', solver='PPL')
[[(0, 0, 1, 0), (0, 0, 1)]]
```

Running the constant-sum solver on a game which isn't a constant sum game generates a `ValueError`:

```
sage: cg = NormalFormGame([A, A])
sage: cg.obtain_nash(algorithm='lp', solver='glpk')
Traceback (most recent call last):
...
ValueError: Input game needs to be a two player constant sum game
```

Here is an example of a 3 by 2 game with 3 Nash equilibrium:

```
sage: A = matrix([[3,3],
....:             [2,5],
....:             [0,6]])
sage: B = matrix([[3,2],
....:             [2,6],
....:             [3,1]])
sage: g = NormalFormGame([A, B])
sage: g.obtain_nash(algorithm='enumeration')
[[(0, 1/3, 2/3), (1/3, 2/3)], [(4/5, 1/5, 0), (2/3, 1/3)], [(1, 0, 0), (1,␣
↪0)]]
```

Of the algorithms implemented, only `'lrs'` and `'enumeration'` are guaranteed to find all Nash equilibria in a game. The solver for constant sum games only ever finds one Nash equilibrium. Although it is possible for the `'LCP'` solver to find all Nash equilibria in some instances, there are instances where it will not be able to find all Nash equilibria.:

```
sage: A = matrix(2, 2)
sage: gg = NormalFormGame([A])
sage: gg.obtain_nash(algorithm='enumeration')
[[(0, 1), (0, 1)], [(0, 1), (1, 0)], [(1, 0), (0, 1)], [(1, 0), (1, 0)]]
sage: gg.obtain_nash(algorithm='lrs') # optional - lrs
[[(0, 1), (0, 1)], [(0, 1), (1, 0)], [(1, 0), (0, 1)], [(1, 0), (1, 0)]]
sage: gg.obtain_nash(algorithm='lp', solver='glpk')
[[(1.0, 0.0), (1.0, 0.0)]]
sage: gg.obtain_nash(algorithm='LCP') # optional - gambit
[[(1.0, 0.0), (1.0, 0.0)]]
sage: gg.obtain_nash(algorithm='enumeration', maximization=False)
[[(0, 1), (0, 1)], [(0, 1), (1, 0)], [(1, 0), (0, 1)], [(1, 0), (1, 0)]]
sage: gg.obtain_nash(algorithm='lrs', maximization=False) # optional - lrs
[[(0, 1), (0, 1)], [(0, 1), (1, 0)], [(1, 0), (0, 1)], [(1, 0), (1, 0)]]
sage: gg.obtain_nash(algorithm='lp', solver='glpk', maximization=False)
[[(1.0, 0.0), (1.0, 0.0)]]
sage: gg.obtain_nash(algorithm='LCP', maximization=False) # optional - gambit
[[(1.0, 0.0), (1.0, 0.0)]]
```

Note that outputs for all algorithms are as lists of lists of tuples and the equilibria have been sorted so that all algorithms give a comparable output (although `'LCP'` returns floats):

```
sage: enumeration_eqs = g.obtain_nash(algorithm='enumeration')
sage: [[type(s) for s in eq] for eq in enumeration_eqs]
[[<... 'tuple'>, <... 'tuple'>], [<... 'tuple'>, <... 'tuple'>], [<... 'tuple
↪'>, <... 'tuple'>]]
sage: lrs_eqs = g.obtain_nash(algorithm='lrs')  # optional - lrslib
sage: [[type(s) for s in eq] for eq in lrs_eqs]  # optional - lrslib
[[<... 'tuple'>, <... 'tuple'>], [<... 'tuple'>, <... 'tuple'>], [<... 'tuple
↪'>, <... 'tuple'>]]
sage: LCP_eqs = g.obtain_nash(algorithm='LCP')  # optional - gambit
sage: [[type(s) for s in eq] for eq in LCP_eqs]  # optional - gambit
```

(continues on next page)

(continued from previous page)

```
[[<... 'tuple'>, <... 'tuple'>], [<... 'tuple'>, <... 'tuple'>], [<... 'tuple
↪'>, <... 'tuple'>]]
sage: enumeration_eqs == sorted(enumeration_eqs)
True
sage: lrs_eqs == sorted(lrs_eqs)  # optional - lrslib
True
sage: LCP_eqs == sorted(LCP_eqs)  # optional - gambit
True
sage: lrs_eqs == enumeration_eqs  # optional - lrslib
True
sage: enumeration_eqs == LCP_eqs  # optional - gambit
False
sage: [[[round(float(p), 6) for p in str] for str in eq] for eq in␣
↪enumeration_eqs] == [[[round(float(p), 6) for p in str] for str in eq] for␣
↪eq in LCP_eqs]  # optional - gambit
True
```

Also, not specifying a valid solver would lead to an error:

```
sage: A = matrix.identity(2)
sage: g = NormalFormGame([A])
sage: g.obtain_nash(algorithm="invalid")
Traceback (most recent call last):
...
ValueError: 'algorithm' should be set to 'enumeration', 'LCP', 'lp' or 'lrs'
sage: g.obtain_nash(algorithm="lp", solver="invalid")
Traceback (most recent call last):
...
ValueError: 'solver' should be set to 'GLPK', ..., None
 (in which case the default one is used), or a callable.
```

payoff_matrices()

Return 2 matrices representing the payoffs for each player.

EXAMPLES:

```
sage: p1 = matrix([[1, 2], [3, 4]])
sage: p2 = matrix([[3, 3], [1, 4]])
sage: g = NormalFormGame([p1, p2])
sage: g.payoff_matrices()
(
[1 2]  [3 3]
[3 4], [1 4]
)
```

If we create a game with 3 players we will not be able to obtain payoff matrices:

```
sage: g = NormalFormGame()
sage: g.add_player(2)  # adding first player with 2 strategies
sage: g.add_player(2)  # adding second player with 2 strategies
sage: g.add_player(2)  # adding third player with 2 strategies
sage: g.payoff_matrices()
Traceback (most recent call last):
...
ValueError: Only available for 2 player games
```

If we do create a two player game but it is not complete then an error is also raised:

```
sage: g = NormalFormGame()
sage: g.add_player(1)  # Adding first player with 1 strategy
sage: g.add_player(1)  # Adding second player with 1 strategy
sage: g.payoff_matrices()
Traceback (most recent call last):
...
ValueError: utilities have not been populated
```

The above creates a 2 player game where each player has a single strategy. Here we populate the strategies and can then view the payoff matrices:

```
sage: g[0, 0] = [1,2]
sage: g.payoff_matrices()
([1], [2])
```

CHAPTER

FOUR

A CATALOG OF NORMAL FORM GAMES.

This allows us to construct common games directly:

```
sage: g = game_theory.normal_form_games.PrisonersDilemma()
sage: g
Prisoners dilemma - Normal Form Game with the following utilities: ...
```

We can then immediately obtain the Nash equilibrium for this game:

```
sage: g.obtain_nash()
[[(0, 1), (0, 1)]]
```

When we test whether the game is actually the one in question, sometimes we will build a dictionary to test it, since the printed representation can be platform-dependent, like so:

```
sage: d = {(0, 0): [-2, -2], (0, 1): [-5, 0], (1, 0): [0, -5], (1, 1): [-4, -4]}
sage: g == d
True
```

The docstrings give an interpretation of each game.

More information is available in the following references:

REFERENCES:

- [?]
- [?]
- [?]
- [?]
- [?]
- [?]

AUTHOR:

- James Campbell and Vince Knight (06-2014)

sage.game_theory.catalog_normal_form_games.**AntiCoordinationGame**(*A=3*, *a=3*, *B=5*, *b=1*, *C=1*, *c=5*, *D=0*, *d=0*)

Return a 2 by 2 AntiCoordination Game.

An anti coordination game is a particular type of game where the pure Nash equilibria is for the players to pick different strategies.

In general these are represented as a normal form game using the following two matrices:

$$A = \begin{pmatrix} A & C \\ B & D \end{pmatrix}$$
$$B = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

Where $A < B, D < C$ and $a < c, d < b$.

An often used version is the following:

$$A = \begin{pmatrix} 3 & 1 \\ 5 & 0 \end{pmatrix}$$
$$B = \begin{pmatrix} 3 & 5 \\ 1 & 0 \end{pmatrix}$$

This is the default version of the game created by this function:

```
sage: g = game_theory.normal_form_games.AntiCoordinationGame()
sage: g
Anti coordination game - Normal Form Game with the following utilities: ...
sage: d ={(0, 1): [1, 5], (1, 0): [5, 1],
....:     (0, 0): [3, 3], (1, 1): [0, 0]}
sage: g == d
True
```

There are two pure Nash equilibria and one mixed:

```
sage: g.obtain_nash()
[[(0, 1), (1, 0)], [(1/3, 2/3), (1/3, 2/3)], [(1, 0), (0, 1)]]
```

We can also pass different values of the input parameters:

```
sage: g = game_theory.normal_form_games.AntiCoordinationGame(A=2, a=3,
....:                                         B=4, b=2, C=2, c=8, D=1, d=0)
sage: g
Anti coordination game - Normal Form Game with the following utilities: ...
sage: d ={(0, 1): [2, 8], (1, 0): [4, 2],
....:     (0, 0): [2, 3], (1, 1): [1, 0]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 1), (1, 0)], [(2/7, 5/7), (1/3, 2/3)], [(1, 0), (0, 1)]]
```

Note that an error is returned if the defining inequality is not obeyed $A > B, D > C$ and $a > c, d > b$:

```
sage: g = game_theory.normal_form_games.AntiCoordinationGame(A=8, a=3,
....:                                         B=4, b=2, C=2, c=8, D=1, d=0)
Traceback (most recent call last):
...
TypeError: the input values for an Anti coordination game must be of the form A <␣
↪B, D < C, a < c and d < b
```

sage.game_theory.catalog_normal_form_games.**BattleOfTheSexes**()
Return a Battle of the Sexes game.

Consider two payers: Amy and Bob. Amy prefers to play video games and Bob prefers to watch a movie. They both however want to spend their evening together. This can be modeled as a normal form game using the

following two matrices [**?**]:

$$A = \begin{pmatrix} 3 & 1 \\ 0 & 2 \end{pmatrix}$$
$$B = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$$

This is a particular type of Coordination Game. There are three Nash equilibria:

1. Amy and Bob both play video games;
2. Amy and Bob both watch a movie;
3. Amy plays video games 75% of the time and Bob watches a movie 75% of the time.

This can be implemented in Sage using the following:

```
sage: g = game_theory.normal_form_games.BattleOfTheSexes()
sage: g
Battle of the sexes - Coordination game -
 Normal Form Game with the following utilities: ...
sage: d = {(0, 1): [1, 1], (1, 0): [0, 0], (0, 0): [3, 2], (1, 1): [2, 3]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 1), (0, 1)], [(3/4, 1/4), (1/4, 3/4)], [(1, 0), (1, 0)]]
```

sage.game_theory.catalog_normal_form_games.**Chicken**(*A=0*, *a=0*, *B=1*, *b=-1*, *C=-1*, *c=1*, *D=-10*, *d=-10*)

Return a Chicken game.

Consider two drivers locked in a fierce battle for pride. They drive towards a cliff and the winner is declared as the last one to swerve. If neither player swerves they will both fall off the cliff.

This can be modeled as a particular type of anti coordination game using the following two matrices:

$$A = \begin{pmatrix} A & C \\ B & D \end{pmatrix}$$
$$B = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

Where $A < B, D < C$ and $a < c, d < b$ but with the extra condition that $A > C$ and $a > b$.

Here are the numeric values used by default [**?**]:

$$A = \begin{pmatrix} 0 & -1 \\ 1 & -10 \end{pmatrix}$$
$$B = \begin{pmatrix} 0 & 1 \\ -1 & -10 \end{pmatrix}$$

There are three Nash equilibria:

1. The second player swerving.
2. The first player swerving.
3. Both players swerving with 1 out of 10 times.

This can be implemented in Sage using the following:

```
sage: g = game_theory.normal_form_games.Chicken()
sage: g
Chicken - Anti coordination game -
 Normal Form Game with the following utilities: ...
sage: d = {(0, 1): [-1, 1], (1, 0): [1, -1],
....:      (0, 0): [0, 0], (1, 1): [-10, -10]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 1), (1, 0)], [(9/10, 1/10), (9/10, 1/10)], [(1, 0), (0, 1)]]
```

Non default values can be passed:

```
sage: g = game_theory.normal_form_games.Chicken(A=0, a=0, B=2,
....:                                 b=-1, C=-1, c=2, D=-100, d=-100)
sage: g
Chicken - Anti coordination game -
 Normal Form Game with the following utilities: ...
sage: d = {(0, 1): [-1, 2], (1, 0): [2, -1],
....:      (0, 0): [0, 0], (1, 1): [-100, -100]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 1), (1, 0)], [(99/101, 2/101), (99/101, 2/101)],
 [(1, 0), (0, 1)]]
```

Note that an error is returned if the defining inequalities are not obeyed $B > A > C > D$ and $c > a > b > d$:

```
sage: g = game_theory.normal_form_games.Chicken(A=8, a=3, B=4, b=2,
....:                                           C=2, c=8, D=1, d=0)
Traceback (most recent call last):
...
TypeError: the input values for a game of chicken must be of the form B > A > C >␣
↪D and c > a > b > d
```

sage.game_theory.catalog_normal_form_games.**CoordinationGame**(*A=10*, *a=5*, *B=0*, *b=0*, *C=0*, *c=0*, *D=5*, *d=10*)

Return a 2 by 2 Coordination Game.

A coordination game is a particular type of game where the pure Nash equilibrium is for the players to pick the same strategies [**?**].

In general these are represented as a normal form game using the following two matrices:

$$A = \begin{pmatrix} A & C \\ B & D \end{pmatrix}$$

$$B = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

Where $A > B, D > C$ and $a > c, d > b$.

An often used version is the following:

$$A = \begin{pmatrix} 10 & 0 \\ 0 & 5 \end{pmatrix}$$

$$B = \begin{pmatrix} 5 & 0 \\ 0 & 10 \end{pmatrix}$$

This is the default version of the game created by this function:

```
sage: g = game_theory.normal_form_games.CoordinationGame()
sage: g
Coordination game - Normal Form Game with the following utilities: ...
sage: d = {(0, 1): [0, 0], (1, 0): [0, 0],
....:      (0, 0): [10, 5], (1, 1): [5, 10]}
sage: g == d
True
```

There are two pure Nash equilibria and one mixed:

```
sage: g.obtain_nash()
[[(0, 1), (0, 1)], [(2/3, 1/3), (1/3, 2/3)], [(1, 0), (1, 0)]]
```

We can also pass different values of the input parameters:

```
sage: g = game_theory.normal_form_games.CoordinationGame(A=9, a=6,
....:                                    B=2, b=1, C=0, c=1, D=4, d=11)
sage: g
Coordination game - Normal Form Game with the following utilities: ...
sage: d ={(0, 1): [0, 1], (1, 0): [2, 1],
....:     (0, 0): [9, 6], (1, 1): [4, 11]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 1), (0, 1)], [(2/3, 1/3), (4/11, 7/11)], [(1, 0), (1, 0)]]
```

Note that an error is returned if the defining inequalities are not obeyed $A > B, D > C$ and $a > c, d > b$:

```
sage: g = game_theory.normal_form_games.CoordinationGame(A=9, a=6,
....:                                   B=0, b=1, C=2, c=10, D=4, d=11)
Traceback (most recent call last):
...
TypeError: the input values for a Coordination game must
                be of the form A > B, D > C, a > c and d > b
```

sage.game_theory.catalog_normal_form_games.**HawkDove**(*v=2*, *c=3*)

Return a Hawk Dove game.

Suppose two birds of prey must share a limited resource v. The birds can act like a hawk or a dove.

- If a dove meets a hawk, the hawk takes the resources.
- If two doves meet they share the resources.
- If two hawks meet, one will win (with equal expectation) and take the resources while the other will suffer a cost of c where $c > v$.

This can be modeled as a normal form game using the following two matrices [**?**]:

$$A = \begin{pmatrix} v/2 - c & v \\ 0 & v/2 \end{pmatrix}$$
$$B = \begin{pmatrix} v/2 - c & 0 \\ v & v/2 \end{pmatrix}$$

Here are the games with the default values of $v = 2$ and $c = 3$.

$$A = \begin{pmatrix} -2 & 2 \\ 0 & 1 \end{pmatrix}$$
$$B = \begin{pmatrix} -2 & 0 \\ 2 & 1 \end{pmatrix}$$

This is a particular example of an anti coordination game. There are three Nash equilibria:

1. One bird acts like a Hawk and the other like a Dove.
2. Both birds mix being a Hawk and a Dove

This can be implemented in Sage using the following:

```
sage: g = game_theory.normal_form_games.HawkDove()
sage: g
Hawk-Dove - Anti coordination game -
 Normal Form Game with the following utilities: ...
sage: d ={(0, 1): [2, 0], (1, 0): [0, 2],
....:     (0, 0): [-2, -2], (1, 1): [1, 1]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 1), (1, 0)], [(1/3, 2/3), (1/3, 2/3)], [(1, 0), (0, 1)]]

sage: g = game_theory.normal_form_games.HawkDove(v=1, c=3)
sage: g
Hawk-Dove - Anti coordination game -
 Normal Form Game with the following utilities: ...
sage: d ={(0, 1): [1, 0], (1, 0): [0, 1],
....:     (0, 0): [-5/2, -5/2], (1, 1): [1/2, 1/2]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 1), (1, 0)], [(1/6, 5/6), (1/6, 5/6)], [(1, 0), (0, 1)]]
```

Note that an error is returned if the defining inequality is not obeyed $c < v$:

sage: g = game_theory.normal_form_games.HawkDove(v=5, c=1) Traceback (most recent call last): ... TypeError: the input values for a Hawk Dove game must be of the form c > v

sage.game_theory.catalog_normal_form_games.**MatchingPennies**()

Return a Matching Pennies game.

Consider two players who can choose to display a coin either Heads facing up or Tails facing up. If both players show the same face then player 1 wins, if not then player 2 wins.

This can be modeled as a zero sum normal form game with the following matrix [**?**]:

$$A = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

There is a single Nash equilibria at which both players randomly (with equal probability) pick heads or tails.

This can be implemented in Sage using the following:

```
sage: g = game_theory.normal_form_games.MatchingPennies()
sage: g
Matching pennies - Normal Form Game with the following utilities: ...
```

(continues on next page)

(continued from previous page)

```
sage: d ={(0, 1): [-1, 1], (1, 0): [-1, 1],
....:     (0, 0): [1, -1], (1, 1): [1, -1]}
sage: g == d
True
sage: g.obtain_nash('enumeration')
[[(1/2, 1/2), (1/2, 1/2)]]
```

sage.game_theory.catalog_normal_form_games.**Pigs**()

Return a Pigs game.

Consider two pigs. One dominant pig and one subservient pig. These pigs share a pen. There is a lever in the pen that delivers 6 units of food but if either pig pushes the lever it will take them a little while to get to the food as well as cost them 1 unit of food. If the dominant pig pushes the lever, the subservient pig has some time to eat two thirds of the food before being pushed out of the way. If the subservient pig pushes the lever, the dominant pig will eat all the food. Finally if both pigs go to push the lever the subservient pig will be able to eat a third of the food (and they will also both lose 1 unit of food).

This can be modeled as a normal form game using the following two matrices [**?**] (we assume that the dominant pig's utilities are given by A):

$$A = \begin{pmatrix} 3 & 1 \\ 6 & 0 \end{pmatrix}$$
$$B = \begin{pmatrix} 1 & 4 \\ -1 & 0 \end{pmatrix}$$

There is a single Nash equilibrium at which the dominant pig pushes the lever and the subservient pig does not.

This can be implemented in Sage using the following:

```
sage: g = game_theory.normal_form_games.Pigs()
sage: g
Pigs - Normal Form Game with the following utilities: ...
sage: d ={(0, 1): [1, 4], (1, 0): [6, -1],
....:     (0, 0): [3, 1], (1, 1): [0, 0]}
sage: g == d
True
sage: g.obtain_nash()
[[(1, 0), (0, 1)]]
```

sage.game_theory.catalog_normal_form_games.**PrisonersDilemma**(*R=-2*, *P=-4*, *S=-5*, *T=0*)

Return a Prisoners dilemma game.

Assume two thieves have been caught by the police and separated for questioning. If both thieves cooperate and do not divulge any information they will each get a short sentence. If one defects he/she is offered a deal while the other thief will get a long sentence. If they both defect they both get a medium length sentence.

This can be modeled as a normal form game using the following two matrices [**?**]:

$$A = \begin{pmatrix} R & S \\ T & P \end{pmatrix}$$
$$B = \begin{pmatrix} R & T \\ S & P \end{pmatrix}$$

Where $T > R > P > S$.

- R denotes the reward received for cooperating.
- S denotes the 'sucker' utility.

- P denotes the utility for punishing the other player.
- T denotes the temptation payoff.

An often used version [**?**] is the following:

$$A = \begin{pmatrix} -2 & -5 \\ 0 & -4 \end{pmatrix}$$
$$B = \begin{pmatrix} -2 & 0 \\ -5 & -4 \end{pmatrix}$$

There is a single Nash equilibrium for this at which both thieves defect. This can be implemented in Sage using the following:

```
sage: g = game_theory.normal_form_games.PrisonersDilemma()
sage: g
Prisoners dilemma - Normal Form Game with the following utilities: ...
sage: d = {(0, 0): [-2, -2], (0, 1): [-5, 0], (1, 0): [0, -5],
....:      (1, 1): [-4, -4]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 1), (0, 1)]]
```

Note that we can pass other values of R, P, S, T:

```
sage: g = game_theory.normal_form_games.PrisonersDilemma(R=-1, P=-2, S=-3, T=0)
sage: g
Prisoners dilemma - Normal Form Game with the following utilities:...
sage: d = {(0, 1): [-3, 0], (1, 0): [0, -3],
....:      (0, 0): [-1, -1], (1, 1): [-2, -2]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 1), (0, 1)]]
```

If we pass values that fail the defining requirement: $T > R > P > S$ we get an error message:

```
sage: g = game_theory.normal_form_games.PrisonersDilemma(R=-1, P=-2, S=0, T=5)
Traceback (most recent call last):
...
TypeError: the input values for a Prisoners Dilemma must be
of the form T > R > P > S
```

sage.game_theory.catalog_normal_form_games.**RPS**()
Return a Rock-Paper-Scissors game.

Rock-Paper-Scissors is a zero sum game usually played between two players where each player simultaneously forms one of three shapes with an outstretched hand.The game has only three possible outcomes other than a tie: a player who decides to play rock will beat another player who has chosen scissors ("rock crushes scissors") but will lose to one who has played paper ("paper covers rock"); a play of paper will lose to a play of scissors ("scissors cut paper"). If both players throw the same shape, the game is tied and is usually immediately replayed to break the tie.

This can be modeled as a zero sum normal form game with the following matrix [**?**]:

$$A = \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix}$$

This can be implemented in Sage using the following:

```
sage: g = game_theory.normal_form_games.RPS()
sage: g
Rock-Paper-Scissors - Normal Form Game with the following utilities: ...
sage: d = {(0, 1): [-1, 1], (1, 2): [-1, 1], (0, 0): [0, 0],
....:      (2, 1): [1, -1], (1, 1): [0, 0], (2, 0): [-1, 1],
....:      (2, 2): [0, 0], (1, 0): [1, -1], (0, 2): [1, -1]}
sage: g == d
True
sage: g.obtain_nash('enumeration')
[[(1/3, 1/3, 1/3), (1/3, 1/3, 1/3)]]
```

sage.game_theory.catalog_normal_form_games.**RPSLS**()

Return a Rock-Paper-Scissors-Lizard-Spock game.

Rock-Paper-Scissors-Lizard-Spock is an extension of Rock-Paper-Scissors. It is a zero sum game usually played between two players where each player simultaneously forms one of three shapes with an outstretched hand. This game became popular after appearing on the television show 'Big Bang Theory'. The rules for the game can be summarised as follows:

- Scissors cuts Paper
- Paper covers Rock
- Rock crushes Lizard
- Lizard poisons Spock
- Spock smashes Scissors
- Scissors decapitates Lizard
- Lizard eats Paper
- Paper disproves Spock
- Spock vaporizes Rock
- (and as it always has) Rock crushes Scissors

This can be modeled as a zero sum normal form game with the following matrix:

$$A = \begin{pmatrix} 0 & -1 & 1 & 1 & -1 \\ 1 & 0 & -1 & -1 & 1 \\ -1 & 1 & 0 & 1 & -1 \\ -1 & 1 & -1 & 0 & 1 \\ 1 & -1 & 1 & -1 & 0 \end{pmatrix}$$

This can be implemented in Sage using the following:

```
sage: g = game_theory.normal_form_games.RPSLS()
sage: g
Rock-Paper-Scissors-Lizard-Spock -
 Normal Form Game with the following utilities: ...
sage: d = {(1, 3): [-1, 1], (3, 0): [-1, 1], (2, 1): [1, -1],
....:      (0, 3): [1, -1], (4, 0): [1, -1], (1, 2): [-1, 1],
....:      (3, 3): [0, 0], (4, 4): [0, 0], (2, 2): [0, 0],
....:      (4, 1): [-1, 1], (1, 1): [0, 0], (3, 2): [-1, 1],
....:      (0, 0): [0, 0], (0, 4): [-1, 1], (1, 4): [1, -1],
....:      (2, 3): [1, -1], (4, 2): [1, -1], (1, 0): [1, -1],
....:      (0, 1): [-1, 1], (3, 1): [1, -1], (2, 4): [-1, 1],
```

(continues on next page)

(continued from previous page)

```
....:       (2, 0): [-1, 1], (4, 3): [-1, 1], (3, 4): [1, -1],
....:       (0, 2): [1, -1]}
sage: g == d
True
sage: g.obtain_nash('enumeration')
[[(1/5, 1/5, 1/5, 1/5, 1/5), (1/5, 1/5, 1/5, 1/5, 1/5)]]
```

sage.game_theory.catalog_normal_form_games.**StagHunt**()

Return a Stag Hunt game.

Assume two friends go out on a hunt. Each can individually choose to hunt a stag or hunt a hare. Each player must choose an action without knowing the choice of the other. If an individual hunts a stag, he must have the cooperation of his partner in order to succeed. An individual can get a hare by himself, but a hare is worth less than a stag.

This can be modeled as a normal form game using the following two matrices [**?**]:

$$A = \begin{pmatrix} 5 & 0 \\ 4 & 2 \end{pmatrix}$$
$$B = \begin{pmatrix} 5 & 4 \\ 0 & 2 \end{pmatrix}$$

This is a particular type of Coordination Game. There are three Nash equilibria:

1. Both friends hunting the stag.
2. Both friends hunting the hare.
3. Both friends hunting the stag 2/3rds of the time.

This can be implemented in Sage using the following:

```
sage: g = game_theory.normal_form_games.StagHunt()
sage: g
Stag hunt - Coordination game -
 Normal Form Game with the following utilities: ...
sage: d = {(0, 1): [0, 4], (1, 0): [4, 0],
....:       (0, 0): [5, 5], (1, 1): [2, 2]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 1), (0, 1)], [(2/3, 1/3), (2/3, 1/3)], [(1, 0), (1, 0)]]
```

sage.game_theory.catalog_normal_form_games.**TravellersDilemma**(*max_value=10*)

Return a Travellers dilemma game.

An airline loses two suitcases belonging to two different travelers. Both suitcases happen to be identical and contain identical antiques. An airline manager tasked to settle the claims of both travelers explains that the airline is liable for a maximum of 10 per suitcase, and in order to determine an honest appraised value of the antiques the manager separates both travelers so they can't confer, and asks them to write down the amount of their value at no less than 2 and no larger than 10. He also tells them that if both write down the same number, he will treat that number as the true dollar value of both suitcases and reimburse both travelers that amount.

However, if one writes down a smaller number than the other, this smaller number will be taken as the true dollar value, and both travelers will receive that amount along with a bonus/malus: 2 extra will be paid to the traveler who wrote down the lower value and a 2 deduction will be taken from the person who wrote down the higher amount. The challenge is: what strategy should both travelers follow to decide the value they should write down?

This can be modeled as a normal form game using the following two matrices [**?**]:

$$A = \begin{pmatrix} 10 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ 11 & 9 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ 10 & 10 & 8 & 5 & 4 & 3 & 2 & 1 & 0 \\ 9 & 9 & 9 & 7 & 4 & 3 & 2 & 1 & 0 \\ 8 & 8 & 8 & 8 & 6 & 3 & 2 & 1 & 0 \\ 7 & 7 & 7 & 7 & 7 & 5 & 2 & 1 & 0 \\ 6 & 6 & 6 & 6 & 6 & 6 & 4 & 1 & 0 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 & 3 & 0 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 2 \end{pmatrix}$$

$$B = \begin{pmatrix} 10 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 \\ 7 & 9 & 10 & 9 & 8 & 7 & 6 & 5 & 4 \\ 6 & 6 & 8 & 9 & 8 & 7 & 6 & 5 & 4 \\ 5 & 5 & 5 & 7 & 8 & 7 & 6 & 5 & 4 \\ 4 & 4 & 4 & 4 & 6 & 7 & 6 & 5 & 4 \\ 3 & 3 & 3 & 3 & 3 & 5 & 6 & 5 & 4 \\ 2 & 2 & 2 & 2 & 2 & 2 & 4 & 5 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 3 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

There is a single Nash equilibrium to this game resulting in both players naming the smallest possible value.

This can be implemented in Sage using the following:

```
sage: g = game_theory.normal_form_games.TravellersDilemma()
sage: g
Travellers dilemma - Normal Form Game with the following utilities: ...
sage: d = {(7, 3): [5, 1], (4, 7): [1, 5], (1, 3): [5, 9],
....:      (4, 8): [0, 4], (3, 0): [9, 5], (2, 8): [0, 4],
....:      (8, 0): [4, 0], (7, 8): [0, 4], (5, 4): [7, 3],
....:      (0, 7): [1, 5], (5, 6): [2, 6], (2, 6): [2, 6],
....:      (1, 6): [2, 6], (5, 1): [7, 3], (3, 7): [1, 5],
....:      (0, 3): [5, 9], (8, 5): [4, 0], (2, 5): [3, 7],
....:      (5, 8): [0, 4], (4, 0): [8, 4], (1, 2): [6, 10],
....:      (7, 4): [5, 1], (6, 4): [6, 2], (3, 3): [7, 7],
....:      (2, 0): [10, 6], (8, 1): [4, 0], (7, 6): [5, 1],
....:      (4, 4): [6, 6], (6, 3): [6, 2], (1, 5): [3, 7],
....:      (8, 8): [2, 2], (7, 2): [5, 1], (3, 6): [2, 6],
....:      (2, 2): [8, 8], (7, 7): [3, 3], (5, 7): [1, 5],
....:      (5, 3): [7, 3], (4, 1): [8, 4], (1, 1): [9, 9],
....:      (2, 7): [1, 5], (3, 2): [9, 5], (0, 0): [10, 10],
....:      (6, 6): [4, 4], (5, 0): [7, 3], (7, 1): [5, 1],
....:      (4, 5): [3, 7], (0, 4): [4, 8], (5, 5): [5, 5],
....:      (1, 4): [4, 8], (6, 0): [6, 2], (7, 5): [5, 1],
....:      (2, 3): [5, 9], (2, 1): [10, 6], (8, 7): [4, 0],
....:      (6, 8): [0, 4], (4, 2): [8, 4], (1, 0): [11, 7],
....:      (0, 8): [0, 4], (6, 5): [6, 2], (3, 5): [3, 7],
....:      (0, 1): [7, 11], (8, 3): [4, 0], (7, 0): [5, 1],
....:      (4, 6): [2, 6], (6, 7): [1, 5], (8, 6): [4, 0],
....:      (5, 2): [7, 3], (6, 1): [6, 2], (3, 1): [9, 5],
....:      (8, 2): [4, 0], (2, 4): [4, 8], (3, 8): [0, 4],
....:      (0, 6): [2, 6], (1, 8): [0, 4], (6, 2): [6, 2],
....:      (4, 3): [8, 4], (1, 7): [1, 5], (0, 5): [3, 7],
....:      (3, 4): [4, 8], (0, 2): [6, 10], (8, 4): [4, 0]}
sage: g == d
True
```

(continues on next page)

(continued from previous page)

```
sage: g.obtain_nash() # optional - lrslib
[[(0, 0, 0, 0, 0, 0, 0, 0, 1), (0, 0, 0, 0, 0, 0, 0, 0, 1)]]
```

Note that this command can be used to create travellers dilemma for a different maximum value of the luggage. Below is an implementation with a maximum value of 5:

```
sage: g = game_theory.normal_form_games.TravellersDilemma(5)
sage: g
Travellers dilemma - Normal Form Game with the following utilities: ...
sage: d = {(0, 1): [2, 6], (1, 2): [1, 5], (3, 2): [4, 0],
....:      (0, 0): [5, 5], (3, 3): [2, 2], (3, 0): [4, 0],
....:      (3, 1): [4, 0], (2, 1): [5, 1], (0, 2): [1, 5],
....:      (2, 0): [5, 1], (1, 3): [0, 4], (2, 3): [0, 4],
....:      (2, 2): [3, 3], (1, 0): [6, 2], (0, 3): [0, 4],
....:      (1, 1): [4, 4]}
sage: g == d
True
sage: g.obtain_nash()
[[(0, 0, 0, 1), (0, 0, 0, 1)]]
```

CHAPTER

FIVE

USING GAMBIT AS A STANDALONE PACKAGE

This file contains some information and tests for the use of Gambit as a stand alone package.

To install gambit as an optional package run (from root of Sage):

```
$ sage -i gambit
```

The python API documentation for gambit shows various examples that can be run easily in IPython. To run the IPython packaged with Sage run (from root of Sage):

```
$ ./sage --ipython
```

Here is an example that constructs the Prisoner's Dilemma:

```
In [1]: import gambit
In [2]: g = gambit.Game.new_table([2,2])
In [3]: g.title = "A prisoner's dilemma game"
In [4]: g.players[0].label = "Alphonse"
In [5]: g.players[1].label = "Gaston"
In [6]: g
Out[6]:
NFG 1 R "A prisoner's dilemma game" { "Alphonse" "Gaston" }

{ { "1" "2" }
{ "1" "2" }
}
""

{
{ "" 0, 0 }
{ "" 0, 0 }
{ "" 0, 0 }
{ "" 0, 0 }
}
1 2 3 4

In [7]: g.players[0].strategies
Out[7]: [<Strategy [0] '1' for player 'Alphonse' in game 'A
prisoner's dilemma game'>,
         <Strategy [1] '2' for player 'Alphonse' in game 'A prisoner's dilemma game'>]
In [8]: len(g.players[0].strategies)
Out[8]: 2

In [9]: g.players[0].strategies[0].label = "Cooperate"
In [10]: g.players[0].strategies[1].label = "Defect"
```

(continues on next page)

(continued from previous page)

```
In [11]: g.players[0].strategies
Out[11]: [<Strategy [0] 'Cooperate' for player 'Alphonse' in game 'A
prisoner's dilemma game'>,
   <Strategy [1] 'Defect' for player 'Alphonse' in game 'A prisoner's dilemma game'>]

In [12]: g[0,0][0] = 8
In [13]: g[0,0][1] = 8
In [14]: g[0,1][0] = 2
In [15]: g[0,1][1] = 10
In [16]: g[1,0][0] = 10
In [17]: g[1,1][1] = 2
In [18]: g[1,0][1] = 2
In [19]: g[1,1][0] = 5
In [20]: g[1,1][1] = 5
```

Here is a list of the various solvers available in gambit:

- ExternalEnumPureSolver
- ExternalEnumMixedSolver
- ExternalLPSolver
- ExternalLCPSolver
- ExternalSimpdivSolver
- ExternalGlobalNewtonSolver
- ExternalEnumPolySolver
- ExternalLyapunovSolver
- ExternalIteratedPolymatrixSolver
- ExternalLogitSolver

Here is how to use the `ExternalEnumPureSolver`:

```
In [21]: solver = gambit.nash.ExternalEnumPureSolver()
In [22]: solver.solve(g)
Out[22]: [<NashProfile for 'A prisoner's dilemma game': [Fraction(0, 1), Fraction(1,␣
↪1), Fraction(0, 1), Fraction(1, 1)]>]
```

Note that the above finds the equilibria by investigating all potential pure pure strategy pairs. This will fail to find all Nash equilibria in certain games. For example here is an implementation of Matching Pennies:

```
In [1]: import gambit
In [2]: g = gambit.Game.new_table([2,2])
In [3]: g[0, 0][0] = 1
In [4]: g[0, 0][1] = -1
In [5]: g[0, 1][0] = -1
In [6]: g[0, 1][1] = 1
In [7]: g[1, 0][0] = -1
In [8]: g[1, 0][1] = 1
In [9]: g[1, 1][0] = 1
In [10]: g[1, 1][1] = -1
In [11]: solver = gambit.nash.ExternalEnumPureSolver()
In [12]: solver.solve(g)
Out[12]: []
```

If we solve this with the LCP solver we get the expected Nash equilibrium:

```
In [13]: solver = gambit.nash.ExternalLCPSolver()
In [14]: solver.solve(g)
Out[14]: [<NashProfile for '': [0.5, 0.5, 0.5, 0.5]>]
```

Note that the above examples only show how to build and find equilibria for two player strategic form games. Gambit supports multiple player games as well as extensive form games: for more details see http://www.gambit-project.org/.

If one really wants to use gambit directly in Sage (without using the NormalFormGame class as a wrapper) then integers must first be converted to Python integers (due to the preparser). Here is an example showing the Battle of the Sexes:

```
sage: import gambit  # optional - gambit
sage: g = gambit.Game.new_table([2,2])  # optional - gambit
sage: g[int(0), int(0)][int(0)] = int(2)  # optional - gambit
sage: g[int(0), int(0)][int(1)] = int(1)  # optional - gambit
sage: g[int(0), int(1)][int(0)] = int(0)  # optional - gambit
sage: g[int(0), int(1)][int(1)] = int(0)  # optional - gambit
sage: g[int(1), int(0)][int(0)] = int(0)  # optional - gambit
sage: g[int(1), int(0)][int(1)] = int(0)  # optional - gambit
sage: g[int(1), int(1)][int(0)] = int(1)  # optional - gambit
sage: g[int(1), int(1)][int(1)] = int(2)  # optional - gambit
sage: solver = gambit.nash.ExternalLCPSolver()  # optional - gambit
sage: solver.solve(g)  # optional - gambit
[<NashProfile for '': [[1.0, 0.0], [1.0, 0.0]]>,
 <NashProfile for '': [[0.6666666667, 0.3333333333], [0.3333333333, 0.6666666667]]>,
 <NashProfile for '': [[0.0, 1.0], [0.0, 1.0]]>]
```

AUTHOR:

- Vince Knight (11-2014): Original version

CHAPTER

SIX

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

g

INDEX

O

P

R

S

T