GAMUT User Guide

Updated June 18, 2004

Current GAMUT Version 1.0.1

1 Introduction

`GAMUT` is a suite of game generators designed for testing game-theoretic algorithms. With `GAMUT`, instances of games from thirty-five base game classes can be easily generated. Using different parameterization options, it is possible to randomize over countless distributions of games, resulting in a comprehensive test bed for any algorithm requiring a normal-form game as input.

This guide is intended to help you get started using `GAMUT`. Section 2 describes how to install and run `GAMUT`. Section 3 describes the `GAMUT` command line and how to use `GAMUT` parameter options. Section 4 lists all of the available game generator, function, and graph classes in the `GAMUT` package, along with more detailed descriptions of the parameter options which can be used to tailor the games generated using these classes. Section 5 introduces the built-in game taxonomy and explains how to use this tool to generate from multiple classes of games which share common properties. Finally, section 6 describes the various formats in which games can currently be output, including the normalization and integer payoff options.

This guide *does not* detail how to develop your own `GAMUT` classes. If you are interested in creating additional games, functions, graphs, or output classes for `GAMUT`, or in modifying the classes that currently exist, refer to the `GAMUT` Developer's Guide, available on our website at http://gamut.stanford.edu.

2 Running `GAMUT`

No installation is required to run `GAMUT`. All that is needed is the file `gamut.jar`. The jar file can be run on the command line using the line

```
java -jar gamut.jar
```
[1]

When this command is typed as given, the `GAMUT` help screen will be displayed, but no games will be generated. In order to generate a game, certain parameters must be specified. This section is meant to give an overview of how to use `GAMUT` parameters on the command line, while details of the particular parameter options available are discussed in more depth in sections 3 and 4.

Some game classes do not require many parameters for generation. One such example is the simple generator for Battle Of the Sexes. An instance of Battle of the Sexes with default payoff values can be generated by adding the -g option to the basic `GAMUT` command line.

```
java -jar gamut.jar -g BattleOfTheSexes
```

Suppose that you would like to generate Battle of the Sexes with payoffs normalized to fall in the range between 0 and 150, and that you would like to store the output in a file called BoS.game. These options can be set using global parameters. Global parameters are specified on the command line in the same manner as the generator class.

[1]A few of the generators may require more memory than is allowed by JVM by default. In such cases it is possible to increase maximum heap size by using `java -Xmx` flag.

```
java -jar gamut.jar -g BattleOfTheSexes -normalize -min_payoff 0 -max_payoff 150
-f BoS.game
```

Other games require more parameters. In order to generate a random game, for example, the number of players and the number of actions for each player must be specified. These parameters may appear on the command line either before, after, or interspersed with the global parameters.

```
java -jar gamut.jar -g RandomGame -players 4 -normalize -min_payoff 0 -max_payoff
150 -f BoS.game -actions 2 4 5 7
```

Finally, some games require the use of function, graph, or subgame classes. Each of these classes must be parameterized as well. When the parameters for a function, graph, or subgame class are specified, they must be enclosed in square brackets. The following command shows how to specify parameters for a random Local-Effect Game using a complete graph and a polynomial function.

```
java -jar gamut.jar -g RandomLEG -players 3 -graph CompleteGraph -graph_params
[-nodes 3 -reflex_ok 0] -func PolyFunction -func_params [-degree 2 -coefs 2 1 3]
```

3 `GAMUT` Command Line Options

The `GAMUT` command line is based on a hierarchy of parameters, each controlling various aspects of the game being generated. Global parameters include the class of games to be used and output options such as settings for normalization. Each class of games then has its own parameters which can be specified. In some cases, subgames, graphs, and functions will be used and must be parameterized as well.

3.1 Global Parameters

The following parameter options are available for all games.

- `-g`: specify which game class or classes to use. If the name specified corresponds to a collection of classes, then a random generator contained in that collection will be chosen. It a list (space separated) of such names is used then the generator will be chosen from an *intersection* of these collections.
- `-random_params`: set this flag if parameters which are not directly set by the user should be randomized. See Section 3.3.
- `-random_seed`: allows user to set a random seed. Defaults to current time. Useful for regenerating the same instance of a "random" game multiple times.
- `-f`: file name for game output.
- `-output`: specify the output class to use. Defaults to SimpleOutput. See Section 6.3.
- `-normalize`: set this flag if payoffs should be normalized within a range. See section 6.1.
- `-min_payoff`: minimum payoff when normalization is used.

- `-max_payoff`: maximum payoff when normalization is used.
- `-int_payoffs`: set this flag if payoffs should be converted to integers rather than output as doubles. See section 6.2.
- `-int_mult`: multiplier used before rounding when converting from double to integer payoffs. Defaults to 10,000.
- `-helpgame`: print help info for a given game (or class of games)
- `-helpgraph`: print help info for given graph class.
- `-helpfunc`: print help info for a given function class.

3.2 Game-Specific Parameters

Each game class in `GAMUT` has its own set of parameters. Two of the most common game-specific parameters, `players` and `actions` are described here. For complete description of the parameters available for use in each game class, see Section 4.

- `-players`: specify the number of players.
- `-actions`: specify the number of actions for each player. May always be entered as one number which will then be the number of actions for each player. In some (non-symmetric) games it is also acceptable to enter the number of actions as a list of action numbers for each player. The version of `actions` used by each individual game class is specified in section 4.1.

3.3 Randomized Parameters

Although some parameters must be set by hand, many parameters can be randomized by setting the `-random_params` flag. When this flag is set, these randomizable parameters may be excluded from the command line and will automatically be set to a random value within some given range. Parameter values that are set by hand take precedence, and don't get randomized even if `-random_params` flag is set.

Often the function, graph, and subgame classes used by a game can be randomized as well. When this is done, the parameters required by these classes will be randomized automatically.

3.3.1 Randomizing Games

It is possible to select a random game from the default distribution if `-g` is ommitted, while `-random_params` is present. If this is the case, a generator will be randomly drawn from the set of generators that accept *both* `-players` and `-actions` parameters, corresponding to the intersection of `GamesWithActionParam` and `GamesWithPlayerParam` classes (see section 5). This feature should be used sparingly, since this default randomization does not include many generators that don't fall into these categories, such as various 2-player or 2-action games or geometric games.

3.4 Default Parameter Settings

Some parameters have default values. These parameters may be excluded from the command line even when the `random_params` flag is not set. When the `random_params` flag is set, these parameters will be randomized, not assigned their default values.

4 Available `GAMUT` Classes

4.1 Games

There are currently thirty-five distinct classes of games available in `GAMUT`. These classes represent games commonly referred to in relevant literature, and are described below.

- `ArmsRace`:

  Create an instance of an Arms Race game. Payoffs in this game are symmetric and calculated by using the formula $-C(x) + B(x - y)$ where x is the level of arms the player in question has chosen, y is the level of arms his opponent has chosen, and C and B are user-specified functions.

  Please note that in order for the game to meet the definition of an Arms Race common in economics literature it must be the case that B is smooth and concave and C is at least smooth. Choose your functions accordingly.

  Game Parameters:

  - `actions`: symmetric version (see section 3.2)
  - `c_func`: the name of the function class to use for the cost function C. The function supplied should be SMOOTH in order to stick to the strict definition of an Arms Race.
  - `c_params`: parameters to be handed off to the cost function C, must be enclosed in [].
  - `b_func`: the name of the function class to use for the B function. The function supplied should be SMOOTH AND CONCAVE in order to stick to the strict definition of an Arms Race.
  - `b_params`: parameters to be handed off to the B function, must be enclosed in [].
  - `low_act`: lower bound on the players' action range. Must be > 0 and ≤ 1000.

- `BattleOfTheSexes`:

  Creates a 2x2 Battle of the Sexes Game

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Game Parameters: No parameters.

- `BertrandOligopoly`:

  Creates an instance of a Bertrand Oligopoly using arbitrary cost and demand functions.

  In the Bertrand Oligopoly, each player offering the object at the lowest price p will receive a payoff of

$p * (D(p)/m) - C(D(p)/m)$

where D is the demand function, C is the cost function, and m is the number of players who offered the object at this price.

Please note that the demand function should be non-negative and decreasing.

Game Parameters:

- `players`
- `actions`: symmetric version (see section 3.2)
- `cost_func`: the name of the function class to use for the cost function.
- `cost_params`: parameters to be handed off to the cost function, must be enclosed in [].
- `demand_func`: the name of the function class to use for the demand function.
- `demand_params`: parameters to be handed off to the demand function, must be enclosed in [].

• `BidirectionalLEG`:

Creates a Bidirectional Local-Effect Game using the specified graph class and specified function class.

A Bidirectional Local-Effect Game is a LEG with a graphical structure in which every edge from b to a has the same local-effect function as the edge from a to b.

Please note that you should be careful when you choose the graph class and set the graph parameters here. The graph chosen should be symmetric (i.e. whenever there is an edge from a to b there is also an edge from b to a) and should not have reflexive edges. (Each node will have a local effect on itself, but this is handled outside of the graph.) Set the parameters for the graph accordingly!

Game Parameters:

- `players`
- `actions`: symmetric version (see section 3.2)
- `graph`: the name of the graph class to use
- `graph_params`: parameters to be handed off to the graph, must be enclosed in [].
- `func`: the name of the function class to use
- `func_params`: parameters to be handed off to the function, must be enclosed in [].

• `Chicken`:

Creates a 2x2 Chicken Game

By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

Game Parameters: No parameters.

- `CollaborationGame`:

  Creates a collaboration game.

  By our definition, both coordination and collaboration games are common payoff yet not always symmetric. The highest payoffs are for all outcomes in which every player chooses the same action. In the collaboration game (unlike in the general coordiation game) these outcomes will all yield the same payoff.

  By default, payoffs will be in the range [-100, 100] with all coordinated payoffs set to 100 for each player. To change this range, use the normalization or integer payoff options.

  Game Parameters:

  – `players`

- `CongestionGame`:

  Creates a congestion game.

  In the congestion game, each player chooses a subset from the set of all facilities. Each player then receives a payoff which is the sum of payoff functions for each facility in the chosen subset. Each payoff function depends only on the number of other players who have chosen the facility.

  Functions used with this generator should always be decreasing in order for the resulting game to meet the criteria for being considered a congestion game.

  Game Parameters:

  – `players`
  – `facilities`: number of facilities in set. Since each player chooses a subset of the facilities, the number of actions available to each player is 2 to the number of facilities. A maximum of five facilities is allowed because of this extremely fast growth in matrix size.
  – `func`: the name of the function class to use for the payoff functions. Should either be a class which always creates decreasing functions, or a class which can be parameterized to create decreasing functions.
  – `func_params`: parameters to be handed off to the function, must be enclosed in []. If the function class in use does not always create decreasing functions, the parameters should be set so that the function is decreasing.
  – `sym_funcs`: should be true if it is desired that all players have the same set of payoff functions.

- `CoordinationGame`:

  Creates a Coordination Game.

  By our definition, coordination games are common payoff yet not always symmetric. The highest payoffs are for all outcomes in which every player chooses the same action, although it is not always the case that all of these outcomes yield the same payoffs. (See collaboration games.)

By default, payoffs will be in the range [-100, 100] with coordinated payoffs positive and uncoordinated payoffs negative. To change this range, use the normalization or integer payoff options.

Game Parameters:

  - `players`

- `CournotDuopoly`:

  Create an instance of the Cournot Duopoly using arbitrary cost and inverse demand functions.

  In order for the problem to make sense, the cost functions used should be increasing. If C_1 and C_2 are cost functions and P is the inverse demand function then if player 1 plays y_1 and player 2 plays y_2, the payoff to player 1 will be $P(y_1+y_2)y_1 - C_1(y_1)$ and the payoff to player 2 will be $P(y_1+y_2)y_2 - C_2(y_2)$

  Although this formulation could be extended to more than two players, this is generally not done in practice so we limit the players to 2.

  Game Parameters:

  - `actions`: symmetric version (see section 3.2)
  - `cost_func1`: the name of the function class to use for the cost function for the first player.
  - `cost_params1`: parameters to be handed off to the cost function for player 1, must be enclosed in [].
  - `cost_func2`: the name of the function class to use for the cost function for the second player.
  - `cost_params2`: parameters to be handed off to the cost function for player 2, must be enclosed in [].
  - `p_func`: the name of the function class to use for the inverse demand function P.
  - `p_params`: parameters to be handed off to the inverse demand function, must be enclosed in [].

- `CovariantGame`:

  Creates a game with the given number of players with payoffs distributed normally(0,1) with covariance r.

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Game Parameters:

  - `players`
  - `actions`: non-symmetric version (see section 3.2)
  - `r`: covariance of any two player's payoffs in the same action profile. Must be between -1/(`players`-1) and 1.

- `DispersionGame`:

  Returns a strong dispersion game which is both action and player symmetric as well as common payoff. An entropy calculation is used in order to determine when one outcome is more dispersed than another, although this could easily be replaced by standard deviation or a similar test.

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Game Parameters:

  - `players`
  - `actions`: symmetric version (see section 3.2)

- `GrabTheDollar`:

  Creates an instance of game Grab the Dollar.

  In this game, there is a prize (or "dollar") that both players are free to grab at any time, where actions represent the chosen times. If both players grab for it at the same time, they will rip the price and both will receive the low payoff. If one chooses a time earlier than the other (i.e. chooses a strictly lower action number number) then he will receive the price (and thus the high payoff) and the opposing player will receive a payoff somewhere between the high and the low.

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Game Parameters:

  - `actions`: symmetric version (see section 3.2) actions in this game. Must be ≥ 1 and ≤ 500, but please note that using large numbers of actions will result in exponentially large games. Large numbers should not be used in games with more than two players.

- `GreedyGame`: Creates a 2 player Greedy Game.

  In this game, each action represents a chosen subset. Player 2 can choose any subset of `set_size` elements while Player 1 can only choose subsets up to size `max_r`.

  If the intersection of the "sets" chosen by the players is empty then the payoff to Player 2 will be the number of elements in the set he has chosen while the payoff to Player 1 will be the negation of this. Otherwise both players will receive 0.

  To change the range of the payoff values, you may use normalization or integer based payoffs.

  Note that the number of actions available to each player is $(\binom{|S|}{maxnumber} + \binom{|S|}{maxnumber-1} + ... + \binom{|S|}{1})$ where *maxnumber* is the maximum number of items in the set that the player can choose.

  Game Parameters:

  - `set_size`: number of elements in set S from which the players choose elements. Must be > 0 and ≤ 8 for the sake of keeping the number of actions reasonable.

  - `max_r`: maximum number of elements which player one (the "red" player) can choose from S. Must be > 0 and $\leq$ `set_size`.

- `GuessTwoThirdsAve`:

  Creates an instance of the game in which all players guess a number trying to come as close as possible to two thirds of the average of the numbers guessed by all players.

  By default, the payoffs for this game are in the range from 0 to 100.0, where the player whose guess comes closest to two thirds of the average receives 100.0 and the others receive 0. If more there is a tie, the payoff amount is split. To change the range of payoffs you can use the normalization or integer payoff options.

  Game Parameters:

  - `players`
  - `actions`: symmetric version (see section 3.2)

- `HawkAndDove`:

  Creates a 2x2 Hawk and Dove.

  Uses the more narrow definition of Hawk and Dove which does not, for example, allow games which would be classified as Prisoners Dilemmas or Chicken Games to qualify as Hawk and Dove.

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Game Parameters: No parameters.

- `LocationGame`:

  Creates an instance of the two person Location Game based on Hotelling's original model.

  In this game there is a street of length l. Player one has a shop set up distance a from one end of the street and player 2 has a shop set up distance b from the other end. Customers are uniformly distributed along the street and the cost of getting a good from a shop to a home on the street is c times the distance. The players must pick a price at which to sell their goods in order to maximize their profit assuming that production is free and customers will always choose the shop for which the combined good price and transportation cost is smaller.

  Profits may be scaled if normalization is used, but relations between the parameters will remain the same and are thus important.

  Be very careful randomizing parameters in this game. If the cost of transporting goods is too high, it will always be a dominant strategy for both players to choose their highest action and the game will lose some of its intended interesting properties.

  Game Parameters:

  - `actions`: symmetric version (see section 3.2)
  - `a`: distance between the location of player 1's store and his end of the street. Must fall between 0 and 1000.

  - `b`: distance between the location of player 2's store and his end of the street. Must fall between 0 and 1000.
  - `l`: length of the entire street. Must be $\geq a + b$ but ≤ 1000.
  - `c`: cost per unit of transporting the goods. Must fall between 1 and 100. See the above note on randomization and values of this parameter.
  - `price_low`: lowest price each player can choose. Must be > 0 and ≤ 1000. The highest price each player can choose will then be `price_low` + `actions` - 1.

- `MajorityVoting`:

  Creates an instance of the Majority Voting Game.

  In this version of the Majority Voting Game, players' utilities for each candidate (i.e. action) being declared the winner are arbitrary and it is possible that a player would be indifferent between two or more candidates.

  If multiple candidates have the same number of votes and this number is higher than the number of votes any other candidate has, then the candidate with the lowest number is declared winner.

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Game Parameters:

  - `players`
  - `actions`: symmetric version (see section 3.2)

- `MatchingPennies`:

  Creates an instance of the Matching Pennies Game

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Game Parameters: No parameters.

- `MinimumEffortGame`:

  Creates an instance of the Minimum Effort Game.

  In this game, the payoff for a player is determined by a formula $a + bM - cE$ where E is the player's effort and M is the minimum effort of any player.

  Game Parameters:

  - `players`
  - `actions`: symmetric version (see section 3.2)
  - `a`: constant a used in formula $a + bM - cE$. Should be between -100 and 100.
  - `b`: coefficient b used in formula $a + bM - cE$. Should be between 0 and 100.
  - `c`: coefficient used in formula $a + bM - cE$. Should be between 0 and 100 but must be $< b$.

- `NPlayerChicken`:

  Creates an instance of the N-Player Chicken Game.

  In N-Player Chicken, just as in the typical two player version of the game, players may cooperate or defect. There is a cost for choosing to cooperate. However, if a certain number of players choose to cooperate, then all players receive a reward.

  The cost and reward amounts are always chosen between 1 and 100 (with reward > cost). To change this range, use normalization.

  Game Parameters:

  - `players`
  - `cutoff`: the number of players who need to cooperate to get the reward. Must be > 0 and $\leq$ `players`.

- `NPlayerPrisonersDilemma`:

  Creates an instance of the N-Player Prisoner's Dilemma Game. In the N-Player Prisoner's Dilemma, the payoff to each player is based on the number of players who cooperate not including the player himself.

  If the number of other players who cooperate is i, then we say that $C(i)$ is the payoff for cooperating and $D(i)$ is the payoff for defecting.In order for this payoff scheme to result in a Prisoner's Dilemma, it must be the case that:

  1) $D(i) > C(i)$ for $0 \leq i \leq n-1$

  2) $D(i+1) > D(i)$ and also $C(i+1) > C(i)$ for $0 \leq i < n-1$

  3) $C(i) > (D(i) + C(i-1))/2$ for $0 < i \leq n-1$

  We guarantee these conditions are met by using linear functions for which you may provide the parameters:

  $C(i) = Xc + Y$

  $D(i) = Xc + Z$

  where $0 < Z - Y < X$.

  Game Parameters:

  - `players`
  - `function_X`: X in payoff functions (see above). Must be set such that $0 < Z - Y < X$ and all parameters must be less than 100,000.
  - `function_Y`: Y in payoff functions (see above). Must be set such that $0 < Z - Y < X$ and all parameters must be less than 100,000.
  - `function_Z`: Z in payoff functions (see above). Must be set such that $0 < Z - Y < X$ and all parameters must be less than 100,000.

- `PolymatrixGame`:

  Creates a polymatrix game using the given graph and the given subgame type to form two player edge games.

If randomization is desired, graph must belong to the `GraphWithNodesParam` class, and subgame must support 2 players and belong to the `GameWithActionParam` class.

Game Parameters:

  - `players`
  - `actions`: symmetric version (see section 3.2)
  - `graph`: the name of the graph structure class to use
  - `graph_params`: parameters to be handed off to the graph, must be enclosed in [].
  - `subgame`: the name of the game class to use as a subgame. There will be an error if the subgame does not have two players or if the number of actions for either of the players is different than that supplied by the actions parameter.
  - `subgame_params`: parameters to be handed off to the subgame, must be enclosed in []. If the `players` or `actions` parameters are generally required by this subgame, they may be left out. These will be reset to appropriate values automatically. All other parameters may be generated randomly and will then be regenerated for each instance of the subgame.

- `PrisonersDilemma`:

  Creates a 2x2 Prisoner's Dilemma

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Game Parameters: No parameters.

- `RandomGame`: Creates a game with the given number of players with payoffs distributed uniformly at random.

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Game Parameters:

  - `players`
  - `actions`: non-symmetric version (see section 3.2)

- `RandomCompoundGame`:

  Creates a Compound Game from a randomly generated symmetric 2x2 matrix.

  A Compound Game is a game in which the payoff for each player is calculated as if he were playing the same two by two game with each of the other players and summing the payoffs.

  The values in the 2x2 game matrix are always chosen at random from values between -100 and 100. To change this range, use the normalization or integer payoff options.

  Game Parameters:

  - `players`

- `RandomLEG`:

  Creates a Local-Effect Game using the specified graph class and specified function class.

  Please note that you should be careful when you choose the graph class and set the graph parameters here. The graph chosen should be symmetric (i.e. whenever there is an edge from a to b there is also an edge from b to a) and should not have reflexive edges. Set the parameters for the graph accordingly!

  Game Parameters:

  – `players`
  – `actions`: symmetric version (see section 3.2)
  – `graph`: the name of the graph class to use
  – `graph_params`: parameters to be handed off to the graph, must be enclosed in [].
  – `func`: the name of the function class to use
  – `func_params`: parameters to be handed off to the function, must be enclosed in [].

- `RandomGraphicalGame`:

  Creates a version of any random graphical game. Parameters for the given graph class must be set. If randomization is desired, graph must belong to the class `GraphWithNodeParam`. Note also that the number of nodes in the graph as implied by graph parameters must match the number of players.

  Game Parameters:

  – `players`
  – `actions`: symmetric version (see section 3.2)
  – `graph`: the name of the graph structure class to use
  – `graph_params`: parameters to be handed off to the graph, must be enclosed in [].

- `RandomZeroSum`:

  Creates a 2 player Zero Sum Game

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Note that when normalization is used, there may be error in the last digits of the decimal payoffs resulting in a games which are occasionally not quite zero sum.

  Game Parameters:

  – `actions`: non-symmetric version (see section 3.2)

- `RockPaperScissors`:

  Creates an instance of the Rock, Paper, Scissors Game.

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Game Parameters: No parameters.

- `ShapleysGame`:

  Creates an instance of Shapley's Game

  By default, all payoffs will fall in the range [0, 100]. This range can be altered by setting the normalization options or by using integer payoffs.

  Game Parameters: No parameters.

- `SimpleInspectionGame`:

  Creates a 2 player Simple Inspection Game

  This game is very similar to the Greedy Game. Each action represents a chosen subset from a set of total size `set_size`. Player 1 can choose any subset of up to `max_r` elements while Player 1 can only choose subsets up to size `max_b`.

  If the intersection of the "sets" chosen by the players is empty then the payoff to Player 2 will be 100.0 while the payoff to Player 1 will be -100.0 Otherwise both players will receive 0.

  To change the range of the payoff values, you may use normalization or integer based payoffs.

  Note that the number of actions available to each player is $(\binom{|S|}{maxnumber} + \binom{|S|}{maxnumber-1} + ... + \binom{|S|}{1})$ where maxnumber is the maximum number of items in the set that the player can choose.

  Game Parameters:

  - `set_size`: number of elements in set S from which the players choose elements. Must be > 0 but ≤ 8 in order o keep the number of actions reasonable.
  - `max_r`: maximum number of elements which player one (the "red" player) can choose from S. Must be > 0 and $\leq$ `set_size`.
  - `max_b`: maximum number of elements which player two (the "blue" player) can choose from S. Must be > 0 and $\leq$ `set_size`.

- `TravelersDilemma`:

  Creates an instance of Traveler's Dilemma game.

  In order to make the game interesting, the parameters should be set up so that the reward is larger than one (but usually smaller than the number of actions). When this holds, the unique Nash equilibrium will be the unsatisfying equilibrium in which everyone chooses the smallest dollar amount.

  When randomization is used, the reward will automatically be chosen from somewhere in this range.

  Game Parameters:

  - `players`
  - `actions`: symmetric version (see section 3.2)
  - `reward`: the amount of the reward for the player who claims the lowest dollar amount. Must be > 0 and ≤ 100.

- `TwoByTwoGame`:

  Creates a game of two actions and two players of a given type according to Rappoport's classification.

  By default, all payoffs will fall in the range [-100, 100]. This range can be altered by setting the normalization options or using integer payoffs.

  Game Parameters:

  - `type`: type of the 2x2 game in Rappoport's classification, in [1,85]

- `UniformLEG`:

  Creates a Uniform Local-Effect Game using the specified graph class and specified function class.

  A Uniform Local-Effect Game is a LEG with a graphical structure in which every edge from b to a has the same local-effect function as the edge from c to a. (This notation is slightly different from what is used in the local-effect literature, but is equivalent.)

  Please note that you should be careful when you choose the graph class and set the graph parameters here. The graph chosen should be symmetric (i.e. whenever there is an edge from a to b there is also an edge from b to a) and should not have reflexive edges. (Each node will have a local effect on itself, but this is handled outside of the graph.) Set the parameters for the graph accordingly!

  Game Parameters:

  - `players`
  - `actions`: symmetric version (see section 3.2)
  - `graph`: the name of the graph class to use
  - `graph_params`: parameters to be handed off to the graph, must be enclosed in [].
  - `func`: the name of the function class to use
  - `func_params`: parameters to be handed off to the function, must be enclosed in [].

- `WarOfAttrition`:

  Creates an instance of the War of Attrition. In a War of Attrition, two players are in a dispute over an object, and each chooses a time to concede the object to the other player. If both concede at the same time, they share the object. Each player has a valuation of the object, and each player's utility is decremented at every time step.

  Payoffs are based on the ranges of valuations and decrements provided. Although normalization may have the effect that the ranges of the payoffs will change, the ratio of the valuation amount to the decrement amount will still come into play.

  Game Parameters:

  - `actions`: symmetric version (see section 3.2)
  - `valuation_low`: lower bound on the players' valuations for the item, should be between 10 and 1000.

- valuation_high: upper bound on the players' valuations for the item. Must be $\geq$ valuation_low.
- decrement_low: lower bound on the amount that the worth of the object to a player is decremented by at each time step. Note that each player has a different decrement value. Should be $\leq$ valuation_low, and between 1 and 100.
- decrement_high: upper bound on the amount that the worth of the object to a player is decremented by at each time step. Must be $\geq$ decrement_low.

4.2 Functions

Game generation often involves the use of functions. The common functions currently built into GAMUT are described below.

- ConcaveTableFunction:

  Represents a general concave function as a table of points. Function is evaluated by looking up nearest point to the x value. No interpolation is done.

  Function Parameters:

  - min: minimum of the function
  - max: maximum of the function
  - points: number of points in the table lookup

- DecreasingWrapper:

  Takes an increasing base function, and makes it descreasing by negating and shifting up.

  Function Parameters:

  - base_func: name of the function to wrap, must be increasing.
  - min: the minimum of the function
  - base_params: parameters to be handed off to the base function, must be enclosed in [].

- ExpFunction:

  A function of the form $f(x) = e^{\alpha * x} + \beta$

  Function Parameters:

  - alpha: multiplicative constant. Should be in the range 0 to 1. Defaults to 1.
  - beta: additive term. Defaults to 0.

- IncreasingPoly:

  Represents an increasing polynomial. Coefficients and degree can either be specified explicitely, or randomized to lie within given ranges.

  Function Parameters:

  - degree: degree of the polynomial function.

  - `coefs`: coefficients of polynomial (should be a list of $d+1$ numbers where d is the degree), in the increasing order of degree.
  - `coef_min`: lower bound on polynomial coefficients, used only for randomizing. Can be anywhere from -1000 to 1000 but defaults to -10.
  - `coef_max`: upper bound on polynomial coefficients, used only for randomizing. Can be anywhere from -1000 to 1000 but defaults to 10.

- `IncreasingTableFunction`:

  Represents a general increasing function as a table of points. Function is evaluated by looking up nearest point to the x value. No interpolation is done.

  Function Parameters:

  - `min`: minimum of the function
  - `max`: maximum of the function
  - `points`: number of points in the table lookup

- `LogFunction`:

  A function of the form $f(x) = \alpha * \ln(x + k) + \beta$. k is calculated automatically as $1 - dMin$, where $dMin$ is the lower bound on the domain.

  Function Parameters:

  - `alpha`: multiplicative constant
  - `beta`: additive term

- `PolyFunction`:

  Represents a general polynomial. Coefficients and degree can either be specified explicitely, or randomized to lie within given ranges.

  Function Parameters:

  - `degree`: degree of the polynomial function.
  - `coefs`: coefficients of polynomial (should be a list of $d+1$ numbers where d is the degree), in the increasing order of degree.
  - `coef_min`: lower bound on polynomial coefficients, used only for randomizing. Can be anywhere from -1000 to 1000 but defaults to -10.
  - `coef_max`: upper bound on polynomial coefficients, used only for randomizing. Can be anywhere from -1000 to 1000 but defaults to 10.

- `TableFunction`:

  Represents a general function as a table of points. Function is evaluated by looking up nearest point to the x value. No interpolation is done.

  Function Parameters:

  - `min`: minimum of the function
  - `max`: maximum of the function
  - `points`: number of points in the table lookup

4.3 Graphs

Game generation can require the use of graphs. Graphical games, local-effect games, and polymatrix games are each generated around the structure of a graph. Some common classes of graphs built into `GAMUT` are described below.

- `BAGraph`:

  Generates a power-law out-degree graph using Barabasi-Albert model. Resulting power-law exponent is around -3.

  Graph Parameters:

  - `m0`: Number of nodes to start with. Defaults to 5.
  - `m`: Number of edges to add to each new node $\leq$ `m0`.
  - `t`: The total number of time steps.

- `CompleteGraph`:

  Generates a complete graph with a specified number of nodes.

  Graph Parameters:

  - `nodes`: Number of nodes in the graph. Must be > 0 and ≤ 100. When randomized, no more than 20 nodes will be added to the graph. For some games, the number should be even less and should be set by hand.
  - `reflex_ok`: Set this to true if reflexive edges are allowed.

- `NAryTree`:

  Generates an n-ary tree with a given branching factor and a given depth.

  Graph Parameters:

  - `n`: Number of children of every non-leaf node in the tree.
  - `depth`: Depth of the tree. Please note that it is advisable to use a very small value for at least one of n and depth parameters to avoid creating graphs too large for the games.

- `NDimensionalGrid`:

  Generates an n-dimensional grid with a given number of points in each dimension. Each node is connected to its neighbors.

  Graph Parameters:

  - `num_dimensions`: Dimensions of the graph. Must be > 0. May be set up to 10, but when randomized will be no greater than 4 since graphs of higher dimensions will be too large for most games.
  - `dim_size`: Size of a single dimension in the graph. Must be > 0. Can be set up to 20, but when randomized will be no greater than 4.

- `NDimensionalWrappedGrid`:

  Generates an n-dimensional grid on a sphere with a given number of points in each dimension. Each node is connected to its neighbors.

  Graph Parameters:

  - `num_dimensions`: Dimensions of the graph. Must be > 0. May be set up to 10, but when randomized will be no greater than 4 since graphs of higher dimensions will be too large for most games.
  - `dim_size`: Size of a single dimension in the graph. Must be > 0. Can be set up to 20, but when randomized will be no greater than 4.

- `PLODGraph`:

  Generates a power-law out-degree graph via PLOD algorithm of Palmer and Stefan.

  Graph Parameters:

  - `nodes`: Number of nodes to generate. Must be ≥ 2 and ≤ 100. Because some games require graphs without too many nodes, no more than 20 nodes will be used when this parameter is randomized. Sometimes it will be necessary to set the parameter to something even smaller by hand.
  - `edges`: The total number of directed/or undirected edges.
  - `alpha`: Alpha parameter (power) in the power law, defaults to 2.1.
  - `beta`: Beta parameter (multiplier) in the power law, defaults to 5.
  - `sym_edges`: Set this to true if it should be the case that whenever there is an edge from node a to node b, there is also an edge from node b to node a.

- `RandomGraph`:

  Generates a (uniformly) random graph according to $G(n, m)$ model.

  Graph Parameters:

  - `nodes`: Number of nodes in the random graph. May be set very large by hand, but when randomized will not be set to anything over 20 since very large graphs do not work well in some games. Occasionally this parameter must be set to something even smaller by hand.
  - `edges`: If `sym_edges` is not set, the total number of directed edges in the random graph. If `sym_edges` is set, the number of pairs of directed edges.
  - `sym_edges`: Set this to true if it should be the case that whenever there is an edge from node a to node b, there is also an edge from node b to node a.
  - `reflex_ok`: Set this to true if reflexive edges are allowed.

- `RingGraph`:

  Generates a ring-of-ring graphs. Consists of a central ring of nodes, each of which participates in a separate outer ring of nodes.

  Graph Parameters:

  - `inner_nodes`: Number of nodes in the inner circle of the ring graph. May be set up to 50 by hand, but when randomized will be set to something no larger than 6 since many games cannot handle large graphs.
  - `outer_nodes`: Number of nodes in each of the outer circles of the ring graph. May be set up to 50 by hand, but when randomized will be set to something no larger than 6 since many games cannot handle large graphs.

- `RoadGraph`:

  Generates a road graph: consists of a two sets of n nodes each connected in a line, with additional n edges connecting corresponding nodes in two sets.

  Graph Parameters:

  - `nodes`: Number of nodes in the road graph. Must be > 0 and ≤ 100. When randomized this parameter will not be set to anything larger than 20 since large graphs do not work well with some games. Occasionally this parameter will need to be set to something even smaller by hand.

- `SmallWorldGraph`:

  Generates a small-world graph according to the Watts-Strogatz model. Starts with a ring lattice of degree 2k, and then randomly rewires each edge with some probability.

  Graph Parameters:

  - `nodes`: Number of nodes in the graph. Must be > 0 and ≤ 100. When randomized, this parameter will not be set to anything over 20 since many games cannot handle large graphs.
  - `K`: Each node will have 2K neighbours in the original ring lattice. Defaults to 2 or can be randomized.
  - `p`: Probability of rewiring each edge

- `StarGraph`:

  Generates a star graph, a single center node connected to all other nodes.

  Graph Parameters:

  - `nodes`: Total number of nodes in the graph. Must be > 0 and ≤ 100. When randomized, this parameter will not be set to anything greater than 20 since many games cannot handle large graphs.

5 The Taxonomy

The taxonomy feature allows generation from classes of games, functions, and graphs that cannot be generated explicitly, but that encompass multiple subclasses that can be explicitly generated. When a class of games is selected using the taxonomy feature, a generator is chosen uniformly at random from the set of all generators which are subclasses of this class, and this randomly chosen generator is invoked to create a game.

5.1 Taxonomy Implementation

The taxonomies with known classes of games, graphs, functions, and outputters are stored in four text files names `games.txt`, `graphs.txt`, `games.txt`, `outputters.txt`, respectively. The default files are stored in the `gamut.jar` under `edu/stanford/multiagent/gamer/` directory.

5.1.1 Overriding File Location

If `GAMUT` is run from a jar file, it will by default try to load taxonomy files stored within the jar file. It is possible to specify an alternative external location for them by setting `gamer.class.path` java property[2]. If the property is not set, and `GAMUT` is not run from jar, it will try to locate the files in the current directory as a last resort.

5.1.2 Taxonomy File Format

Taxonomy files consist of class name followed by the equal sign `=` followed by class definition. Empty lines and lines starting with `#` are ignored. Class definition may take one of three forms:

- **=Empty Line**: This indicates a basic *ground* generator which can be instantiated and invoked to generate a game, graph, function, or an outputter.
- **= [ClassName -param value -param value** ...]: This is also a ground class, with *partially specified parameters.* Here `ClassName` must be a basic generator name. When instantiating such a class, some parameters will be preset. This is useful in order to split up ground generators into finer subclasses.
- **= Class1, Class2, Class3,** ...: This specifies a *collection* of classes. To instantiate such a class, `GAMUT` will pick a subclass from the list at random, and recursively repeat this until it gets to a ground generator that can be executed.

5.2 Game Classes

The following classes of games are formed by randomizing over appropriate subclasses.

- `GameWithActionParam`: The class of games that are action-extensible. This class can be used, for example, as the subclass for polymatrix games.

  Component classes: GameWithActionParam= MajorityVoting, TravelersDilemma, LocationGame, PolymatrixGame, RandomZeroSum, BertrandOligopoly, DispersionGame, RandomGraphicalGame, CournotDuopoly, BidirectionalLEG, RandomLEG, GuessTwoThirdsAve, UniformLEG, CovariantGame, GrabTheDollar, WarOfAttrition, RandomGame, MinimumEffortGame

- `GameWithPlayerParam`: The class of games that are player-extensible in a nice way.

  Component classes: BertrandOligopoly, BidirectionalLEG, CollaborationGame, CongestionGame, CoordinationGame, CovariantGame, DispersionGame, GuessTwoThirdsAve, MajorityVoting, MinimumEffortGame, NPlayerChicken, NPlayerPrisonersDilemma, PolymatrixGame, RandomGame, RandomCompoundGame, RandomLEG, RandomGraphicalGame, TravelersDilemma, UniformLEG

[2]This can be done using `java -Dgamer.class.path=dirname` flag

- `Game2PlayerOrParam`: Similar to above but for two player or two action games.

  Component classes: GameWithPlayerParam, BattleOfTheSexes, Chicken, CournotDuopoly, GrabTheDollar, HawkAndDove, LocationGame, MatchingPennies, PrisonersDilemma, RandomZeroSum, RockPaperScissors, TwoByTwoGame, WarOfAttrition

- `Game2ActionOrParam`

  Component classes: GameWithActionParam, BattleOfTheSexes, CollaborationGame, CoordinationGame, HawkAndDove, MatchingPennies, NPlayerChicken, NPlayerPrisonersDilemma, PrisonersDilemma, RandomCompoundGame, TwoByTwoGame

- `OriginalPaperNPlayerDist`

  Component classes: DispersionGame, MinimumEffortGame, RandomGame, TravelersDilemma, BertrandOligopoly, PolymatrixGame-SW, PolymatrixGame-RG, PolymatrixGame-Road, PolymatrixGame-CG, UniformLEG-RG, UniformLEG-CG, UniformLEG-SG, BidirectionalLEG-RG, BidirectionalLEG-CG, BidirectionalLEG-SG, GraphicalGame-RG, GraphicalGame-SG, GraphicalGame-Road, GraphicalGame-SW, CovariantGame-Pos, CovariantGame-Zero, CovariantGame-Rand

- `OritinalPaper2PlayerDist`

  Component classes: OriginalPaperNPlayerDist, LocationGame, WarOfAttrition, CovariantGame-Neg

- `SymmetricTwoByTwo`

  Component classes: BattleOfTheSexes, Chicken, HawkAndDove, PrisonersDilemma

- `ClassicalMatrixgame`

  Component classes: BattleOfTheSexes, MatchingPennies, PrisonersDilemma, HawkAndDove, Chicken, RockPaperScissors, CollaborationGame, CoordinationGame

- `CompactlyRepresentable`

  Component classes: BidirectionalLEG, PolymatrixGame, UniformLEG, RandomLEG, CoordinationGame, GraphicalGame-Road

- `CompleteOpposition`

  Component classes: RockPaperScissors, MatchingPennies, SimpleInspectionGame, GreedyGame, RandomZeroSum, GeometricGame

- `CompoundGame`

  Component classes: NPlayerChicken, NPlayerPrisonersDilemma, RandomCompoundGame

- `CongestionGameClass`

  Component classes: BidirectionalLEG, UniformLEG, DispersionGame, CongestionGame

- `CoordinationGameClass`

  Component classes: MinimumEffortGame, CollaborationGame, BattleOfTheSexes

- `DominanceSolvableEq`

  Component classes: TravelersDilemma, PrisonersDilemma, NPlayerPrisonersDilemma, SupermodularGames

- `DominantStrategies`

  Component classes: PrisonersDilemma, TravelersDilemma, NPlayerPrisonersDilemma

- `ESSGames`

  Component classes: SymmetricTwoByTwo

- `GeometricGame`

  Component classes: MatchingPennies, GreedyGame, SimpleInspectionGame

- `NoDominantStrategies`

  Component classes: MatchingPennies, BattleOfTheSexes, DispersionGame, RockPaperScissors

- `NoPSNE`

  Component classes: MatchingPennies, SimpleInspectionGame, RockPaperScissors

- `PotentialGameClass`

  Component classes: CongestionGameClass

- `PSNEGameClass`

  Component classes: CoordinationGameClass, PotentialGameClass, DominanceSolvableEq, BertrandOligopoly, Chicken, ArmsRace, CournotDuopoly

- `StrictEqGameClass`

  Component classes: NPlayerPrisonersDilemma, HawkAndDove, PrisonersDilemma, SupermodularGames, LocationGame, MatchingPennies

- `SupermodularGames`

  Component classes: CournotDuopoly, BertrandOligopoly, ArmsRace

- `StronglySymmetricGames`

  Component classes: ESSGames, UniformLEG, BidirectionalLEG, Chicken, SymmetricTwoByTwo, CompoundGame, ArmsRace, BattleOfTheSexes, NPlayerPrisonersDilemma, MatchingPennies, NPlayerChicken, PrisonersDilemma, RandomLEG

- `WeaklySymmetricGames`

  Component classes: StronglySymmetricGames

- `UniqueNEGames`

  Component classes: DominanceSolvableEq, GuessTwoThirdsAve

The next set of classes are formed by creating instances of other classes with particular parameter settings.

- `PolymatrixGame-SW`:

  [PolymatrixGame -graph SmallWorldGraph -subgame RandomGame]

- `PolymatrixGame-RG`:

  [PolymatrixGame -graph RandomGraph -subgame RandomGame -graph_params [-sym_edges]]

- `PolymatrixGame-Road`:

  [PolymatrixGame -graph RoadGraph -subgame RandomGame]

- `PolymatrixGame-CG`:

  [PolymatrixGame -graph CompleteGraph -subgame RandomGame]

- `UniformLEG-RG`:

  [UniformLEG -graph RandomGraph -graph_params [-reflex_ok 0 -sym_edges]]

- `UniformLEG-CG`:

  [UniformLEG -graph CompleteGraph -graph_params [-reflex_ok 0 -sym_edges]]

- `UniformLEG-SG`:

  [UniformLEG -graph StarGraph -graph_params [-reflex_ok 0 -sym_edges]]

- `BidirectionalLEG-RG`:

  [BidirectionalLEG -graph RandomGraph -graph_params [-reflex_ok 0 -sym_edges]]

- `BidirectionalLEG-CG`:

  [BidirectionalLEG -graph CompleteGraph -graph_params [-reflex_ok 0 -sym_edges]]

- `BidirectionalLEG-SG`:

  [BidirectionalLEG -graph StarGraph -graph_params [-reflex_ok 0 -sym_edges]]

- `GraphicalGame-RG`:

  [RandomGraphicalGame -graph RandomGraph -graph_params [-reflex_ok 0 -sym_edges]]

- `GraphicalGame-SG`:

  [RandomGraphicalGame -graph StarGraph -graph_params [-reflex_ok 0 -sym_edges]]

- `GraphicalGame-Road`:

  [RandomGraphicalGame -graph RoadGraph -graph_params [-reflex_ok 0 -sym_edges]]

- `GraphicalGame-SW`:

  [RandomGraphicalGame -graph SmallWorldGraph -graph_params [-reflex_ok 0 -sym_edges]]

- `CovariantGame-Neg`:

  [CovariantGame -r -0.9]

- `CovariantGame-Pos`:

  [CovariantGame -r 0.9]

- `CovariantGame-Zero`:

  [CovariantGame -r 0]

5.3 Graph Classes

There is currently one graph class available which is created by randomizing over existing graph classes.

- `GraphWithNodeParam`: graph classes which accept the `node` parameter. These can be used with graphical games and LEGs.

  Component classes: CompleteGraph, RandomGraph, RoadGraph, StarGraph, SmallWorld-Graph

5.4 Function Classes

The following classes are created by randomizing over function classes.

- `SlowlyIncreasingFunction`

  Component classes: LogFunction, IncreasingPoly

- `ConcaveFunction`

  Component classes: LogFunction, ConcaveTableFunction

- `IncreasingFunction`

  Component classes: ExpFunction, LogFunction, IncreasingPoly, IncreasingTableFunction

- `DecreasingFunction`

  Component classes: DecreasingWrapper

6 Output

Currently there are no additional output classes available as part of the taxonomy.

6.1 Normalization

Payoffs can be normalized to fall within a given range by setting the `-normalize` flag. When this flag is set, the additional parameters `-min_payoff` and `-max_payoff` must also be set. These parameters specify the values of the lowest and highest payoffs in the game. They parameters do not have default values and cannot be randomized.

Note that payoffs are not normalized for each individual player, but are normalized over all players. Thus the `min_payoff` will be the lowest payoff to any player over all outcomes and similarly the `max_payoff` will be the highest payoff to any player over all outcomes. All other payoffs will be scaled proportionally to fall within this range.

6.2 Integer Versus Double Payoff Values

By default, payoffs for most games will be output as double values. Payoffs can be converted to integer output values by setting the `-int_payoffs` flag. Integer payoff values are calculated by multiplying each payoff by some large constant set using the `-int_mult` parameter and then rounding.

Note that normalization is applied before payoffs are converted to integer values. This implies, for example, that if payoffs are normalized to fall in the range between, say, -100 and 100, and integer payoffs are requested with an `int_mult` of 1000, then payoffs will actually fall on integer values in the range between -100,000 an 100,000.

6.3 Available Output Classes

The following output formats are currently available in `GAMUT`.

- `SimpleOutput`: (default)

  The default output format lists all payoffs for a given output on one line. For example, in a two-player game, the line

  [2 3] : [799169 148480]

  would signify that when player 1 plays his second action and player 2 plays her third action, player one receives a payoff of 799,169 and player 2 receives a payoff of 148,480.

  In this file format, comments precede the payoff values. Lines containing comments begin with #.

- `GambitOutput`:

  Outputs the game in Gambit's .nfg format. More information on this .nfg format can be found on the Gambit website at http://econweb.tamu.edu/gambit/.

  In this format, comments follow the initial line, but precede payoff values.

- `GTOutput`:

  Outputs the game in a format which can be read in by GameTracer. For more on the GT input format, see the GameTracer website at http://dags.stanford.edu/Games/gametracer.html.

  There are no comments in this format.

- `TwoPlayerOutput`:

  This format is designed to resemble the normal matrix form representation of a two player game and thus be human-readable. Payoffs for each outcome appear in pairs in which the first number is the payoff to player 1 and the second number is the payoff to player 2. Each row represents an action choice for player 1, and each column an action choice for player 2.

In this file format, comments precede the payoff values. Lines containing comments begin with two forward slashes.

This formatting option can only be applied to two-player games.

- `SpecialOutput`:

  The `SpecialOutput` class is used for game-specific output formats. Currenly the only game-specific output format available is for graphical games.

7 Additional References

The most up-to-date source code, developer's guide, database of games and references, relevant papers, and contact information can be found on the `GAMUT` website at http://gamut.stanford.edu. Questions or comments on this guide may be sent to jwortman@cs.stanford.edu.