Bluff

Reference manual for Bluff 0.1.0

Paolo Gatti <paolog@users.sourceforge.net>

Table of Contents

1 Introduction 1

2 Copying 2

3 Game rules 3

4 Game design 5

5 Mathematical model 6

6 Extending the game 8

1 Introduction

Bluff is a dice game based both on luck and on ability. It can be played by two or more players, although the best range is probably between three and six. It was begun as an experiment in AI and in game design.

Bluff consists of a backend, which implements the rules of the game, and in several front-ends, which interface the game to its human or computer-controlled players.

At the moment, only two front-ends are available:

- A single text window version, which can be used by a single player only;
- A network version, based on the MUSES library, which starts up a multiplayer server to which one can connect via telnet. The MUSES library is free software and can be found at `http://muses.sourceforge.net`

Hopefully, someone else will write a graphical front-end in the future. The original author is not likely to code one.

The rest of this manual describes the rules of the game, the reasons that led to the choice of those rules, a simplified mathematical model of the game, and some guidelines to follow if you are interested in writing a new front-end or a new kind of computer-controlled player.

2 Copying

3 Game rules

Bluff is a game for two or more players; each of them has a set of five 6-sided dice, some of which are visible to the other players, and some of which are hidden - only their owner can watch them.

Dice combinations have different values: in ascending order,

- Nothing: (1, 2, 3, 5, 6) or (1, 2, 4, 5, 6).
- A single pair
- Two pairs
- Three of a kind
- Small straight: (1, 2, 3, 4, X), (2, 3, 4, 5, X) or (3, 4, 5, 6, X)
- Full house: three of a kind and two of another
- Four of a kind
- Large straight: (1, 2, 3, 4, 5) or (2, 3, 4, 5, 6)
- Five of a kind

Within each category, the highest-valued dice have priority. Three two's are better than three ones; three two's and a six are better than three two's and a four.

At the beginning of the game, each player rolls all of the dice and keeps the resulting combination hidden. Then, each player declares a score; this score might match the actual dice rolled, might be underestimated, or might be a bluff.

Whoever declares the highest score receives one point from all of the other players; the second highest bidder receives one point from lower score bidders; and so on, down to the player who declared the lowest score, who pays one point to every other player in the game.

At this point, each player has to decide whether to trust other players' bids, or to call someone else's bluff. If player A declares that player B bluffed, and player B was not bluffing, A pays one point to B. If B was bluffing, on the other hand, two points are transferred from B to A.

Declaring a wildly high score, therefore, is not a wise strategy: one will gain one point per player in the first phase, but will lose twice the amount when every other player will call the bluff.

Never bluffing leads to somewhat better results; however, it is not the optimal choice, as will be proven with a simplified model.

The best strategy consists in bluffing only some of the times: the opponents will be confused into calling a bluff when one is not bluffing, and vice versa.

After every player has decided which bids to trust, the computer will reveal to everyone which players were bluffing and which players were not bluffing. The actual values of the dice, however, are not disclosed.

Each player can now reroll any amount of dice (none, all, or some). The number of rerolled dice is notified to everyone; the actual rerolled dice are kept private. Then, every player has to reveal the values of two dice to the other players; the choice of which dice is up to the player.

At this point, the game starts anew, with the added quirk that some of the dice are visible. There is another turn of bidding and another turn of trusting/not trusting other players' bids.

After the second hand of the game is over, the reroll-show-bid-trust process is repeated once more. This time, however, four out of five dice are visible: each player gets to keep only one die hidden.

Once the third hand is complete, the game is over. It is possible to restart from the beginning, rerolling all the dice.

4 Game design

Bluff was designed with the following requirements.

1. Bluff should be playable on a network, between untrusted players who might resort to cheating. To discourage this, any alliance between two players must not be able to damage players outside the alliance: the total score of any alliance must not exceed, on average, the total score the same players would achieve playing separately.
2. Trivial strategies, such as never bluffing or always bluffing, should not be optimal. To be true to its name, Bluff should encourage a reasonable amount of bluffing.
3. There should be a balance between the skill and luck elements.
4. The minimum duration of a game should not be too long, not to bore players.

While the second and third requirement are not too hard to satisfy, the first one rules out many game variants. If players are not to gain an advantage by communicating in secret with each other, most information must be made public, and each player's private information must not bear any correlation to any other player's.

Thus, Bluff uses dice (which are rolled independently of each other), rather than cards (which are unique in a deck: knowing N-1 people's cards can reveal fundamental information on the N'th player's hand). Rather than passing around a single set of dice, each player uses a different set: this reduces the amount of secret knowledge that cheaters might want to share behind the scenes. Lastly, as much information as possible (amount of dice rolled, bids, who trusts whose bid...) is disclosed to all players on the same footing.

The first phase of scoring (give points to the highest bidder) is meant to satisfy half of the second requirement, by encouraging players to bluff. The second phase (trusting/not trusting other players), on the other hand, makes overbluffing a dangerous habit, especially when other players catch on.

After the trust phase is over, the list of people who bluffed in the current turn is made public, regardless of who trusted whom. This is again a consequence of the first requirement. If bluff information were only revealed to those who call the bluff, players might share this information behind the scenes; if bluffs were only revealed when someone takes a risk and calls them, alliances could agree on a single player to act as "guinea pig" – thus gaining an advantage against outsiders.

The choice of scoring is open to tuning. The only requirement is that the penalty for a discovered bluff must be higher than the gain for a high bid. When the discovered bluff penalty becoms higher, players are encouraged to call bluffs more lightly, and are discouraged from bluffing. Increasing the cost of calling a bluff when there was none has the opposite effect.

The choice of repeating the game several times, with an increasing amount of dice visible, is due to the third and fourth requirements. Players have to estimate the probability of reaching a certain score starting from certain conditions; this increases the element of skill in the game.

Limiting the game to three turns, instead of five as could seem more natural, is another skill-vs-luck decision. In very long games, lucky players will achieve a high score (say, a full or better) in an early hand; they would then keep this score throughout the match, earning many points from a single lucky dice roll. A three-turn game reduces this problem, and also decreases the chances of the game becoming boring.

5 Mathematical model

This chapter will examine a very simplified model of Bluff, in order to prove that never bluffing is not an optimal strategy.

The real Bluff game allows a wide range of bids, from nothing to five sixes. There are 252 possible bids, which would not be practical to examine here. Suppose instead the dice can only give three results: in ascending order of value, A, B, and C. All these results are equiprobable; a match consists of a single hand.

Player 1 will adopt the following bidding strategy:

- If the actual result is A,
  - bluff and bid C with probability c;
  - bluff and bid B with probability b;
  - don't bluff and bid A with probability 1 - b - c;
- If the actual result is B,
  - bluff and bid C with probability c';
  - don't bluff and bid B with probability 1 - c';
- If the actual result is C, always bid C.

Player 2, on the other hand, will never bluff.

As for the trusting strategy, player 1 will never call a bluff, since player 2 never bluffs. As for player 2, the most generic trust strategy is:

- Always trust a bid of A (being the lowmost score, A is never a bluff);
- If the opponent bids B, and my bid is X, call a bluff with probability pX (pA, pB or pC);
- If the opponent bids C, and my bid is X, call a bluff with probability qX (qA, qB or qC)

Player 2's trust strategy is thus determined by six numbers.

Suppose now that a large number of matches is played. For each possible situation (player 1 score, player 1 bid, player 2 score), what is the expected score?

For example, if player 1 has an A, but bluffed and bid a C, while player 2 has a B, then player 2 will lose one point in any case, because player 1 bid a higher score; with probability qB, player 2 will call the bluff, and gain two points. The expected score for player 2 in this situation is therefore 2qB-1.

Computing the same result for all the possible situations, one obtains the following table (where probabilities are relative, rather than absolute):

Player 1 score	A	A	A	B	B	C
Player 1 bid	A	B	C	B	C	C
Probability	1-b-c	b	c	1-c'	c'	1
Player 2 score						
A	0	2pA-1	2qA-1	-pA-1	2qA-1	-qA-1
B	1	2pB	2qB-1	-pB	2qB-1	-qB-1
C	1	2pC+1	2qC	-pC+1	2qC	-qC

The expected score for player 2 over nine matches is just the sum of the numbers in the lower half of the table, each weighted by the number in the "probability" row.

One can immediately notice that this average score is a sum of seven contributions: some terms are independent of the pX and qX parameters, while some others are respectively proportional to pA, pB, pC, qA, qB and qC. There are no quadratic or higher-order terms in the parameters of player 2's strategy; therefore, maximizing player 2's score is achieved by maximizing each contribution separately.

For example, the term proportional to pA is the following:

pA (2b - 1 + c')

If 2b + c' > 1, this term is maximized by setting pA = 1 (the maximum allowed value); if 2b + c' < 1, player 2 had better set pA = 0 instead.

By repeating the same reasoning for all six coefficients, player 2 obtains the following recipe:

- Set pA, pB, pC = 1 if 2b + c' > 1; set them to 0 otherwise.
- Set qA, qB, qC = 1 if 2c + 2c' > 1; set them to 0 otherwise.

If player 1 decides to operate at threshold, with 2b + c' = 2c + 2c' = 1, player 2's score will be the same whatever values are chosen for pX and qX. By calculating it in the case pX = qX = 0, player 2's score turns out to be

```
S = 2 (1-b-c) -2 c -2 c' - 2 =
  = -2 (b + 2c + c') =
  = -2 ((b + c'/2) + (2 c + 2 c') - 3/2 c') =
  = -2 (1/2 + 1 - 3/2 c') =
  = 3 (c' - 1)
```

The best threshold strategy for player 1 consists in putting c' = 0, and hence b = c = 1/2. When player 1 rolls an A, he always bluffs and declares B or C with even probability; when a B or C is rolled, player 1 does not bluff.

Whatever strategy player two adopts, it will lead to an average loss of at least three points every nine matches.

This concludes the proof that, at least in this simplified model, never bluffing is not an optimal strategy. A similar result can be reasonably expected to hold true even in the more complicated case of the complete Bluff rules.

6 Extending the game

In order to write a new front-end for Bluff, to extend an existing one, or to add a new AI engine, some guidelines should be followed.

Bluff represents players, either human or computer-controlled, via descendants of the abstract base class `Player`. A concrete `Player` must implement several operations described below; each method takes a `GameView` as parameter, and is expected to interact with the game only via the `GameView`'s public interface.

In a multiplayer game, `Player` objects must not block the program flow. This means that methods must either take a decision immediately, or cache the `GameView` and use it at a later time, when the decision has finally been taken.

The `Player` interface consists of the following methods.

- `void start(GameView& view)`: notify the player that a new game is about to start. It is possible to cache *view* for future use.
- `void reroll(GameView& view)`: ask the player to decide which dice (if any) to reroll. Either instantly or at a later time, `view.reroll()` should be called to inform the game of the decision.
- `void cover(GameView& view, unsigned int n)`: ask the player to choose n dice to be hidden from other players. Either instantly or afterwards, use `view.cover()` to commit the list of covered dice.
- `void bid(GameView& view)`: ask the player's bid, which should be delivered via `view.bid()`.
- `void trust(GameView& view)`: ask the player to loop over the other players, starting from number 1 and proceeding in order up to player number `view.numPlayers() - 1`. Each player's bid should be displayed, and a decision to trust or call a bluff taken via `view.trust()`.
- `void wait(GameView& view)`: player should wait till the others have completed their turn.
- `void endTurn(GameView& view)`: display the outcome of the turn that has just been completed.
- `void endGame(GameView& view)`: the game has ended, *view* is about to become invalid and should not be referenced any longer once `endGame()` returns.

The `GameView` can be used to retrieve any kind of information about the game status, provided it is available to the current player. So, a player can inquire about his own dice in the current or previous hands, or about who trusted whom in a previous hand, but not about some other player's covered dice.

The `GameView` is also used to tell the game about the player's decisions, via the methods listed above: `reroll()`, `cover()`, `bid()` and `trust()`. All four methods return true on success, and false if the game rules are being violated (say, someone tries to play out of turn).

If the new type of `Player` is controlled by the computer, it should be registered with the global `AIFactory`. This does not require any modification to existing code: it is sufficient to add a registration line to the AI module, and link it to the game.

Supposing the new AI `Player` class is called `BogoPlayer`, this snippet of code will do the trick:

```
#include "AIFactory.hh"
static AIFactory::type_declaration<BogoPlayer> bogus
("bogo",        // Unique identifier for this AI type
 "a short description (about 50 characters or less)",
 "Here should go a more complete description of the AI type.\n"
 "It can span multiple \\n-terminated lines.\n"
);
```

New AI modules should generally be placed in the `ai/` directory; the existing `Makefile` should handle them appropriately. Reusable AI tools should instead go in `tools/`.

When writing a new frontend, the game loop should work along these lines:

- Create a new `Game`, *myGame*
- Add players to *myGame*
- When everyone is added, call `myGame.start()`
- Then, as long as `myGame.status() != Game::END_GAME`,
  - If there are any asynchronous players (human players in a multiplayer game, computer-controlled players running in a different process...), collect their input and forward it to the appropriate `GameView` objects;
  - Call `myGame.update()`: if all the players have completed a phase of the game, the game will advance to the next phase and players will be prompted.
- Clean up. `Player` objects should remain alive at least as long as the old `Game` is available.

The game advances to the next phase only during `update()`, and not immediately after the last player makes the expected move, in order to spare the `Player` coder some consistency checks. If the game were updated instantly, a `Player`'s `GameView` could be invalidated within a call to `trust()`, because the game reached its conclusion. With a separate `update()` phase, *view* is guaranteed to remain valid whatever the `Player` does.