Chowka Bhara: A Game of Dice

CS 6601 Mini Paper

I. Problem Addressed

To implement an optimal strategy for a computer player in this partly stochastic, partly strategic "Game of Dice". The game, designed for 2-4 players, can be played on a square 5x5, 7x7 and 9x9 board, 'X' marking safe houses, with each player having a fixed set of pawns and different starting points (marked in colors). A player wins if he moves all of his pawns to the center square along a predefined path (Fig. 1).

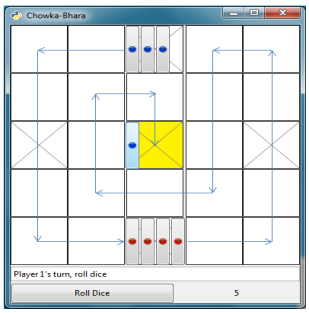

Fig. 1: GUI showing the predefined path for Player Blue

The game has several intricate rules which we have listed out in Appendix C.

The game poses interesting challenges that need to be addressed through multiple heuristics and ranked in order of their relative importance. There is a considerable amount of work done in analyzing popular partially stochastic games like Backgammon. The objective of our project is to build upon such related work, bring it under the purview of AI scientists and revive this ancient Indian game.

II. Related Work

There are several variations of the game online without an AI component; the only existing computer player implementation[1] uses a weighted tree mechanism as an evaluation strategy to rank each of the moves. However, the factors considered in weighing each move are relatively few and do not vary with time (we address it in the next section). While these implementations are interesting, they require multiple human players, have a single user interface and fail to provide challenging experiences at multiple levels of difficulty.

There are some popular adversarial search algorithms which we plan to explore in building the AI for our game. Minimax [2] algorithm assumes that both players play an optimal game. Max will prefer to move to a state of maximum value, whereas Min prefers a state of minimum value. This definition of optimal play for Max assumes that Min also plays optimally-it maximizes the worst-case outcome for Max.

Expectiminimax [3] is an extention of Minimax that takes into account chance nodes. The evaluation function at each chance node is evaluated as Sum(P(s) * Expectiminimax:(S)) where S is the successor to the chance node and can either be a Min or Max node and P(S) is the probability associated with the successor.

Chanceprobcut [4] is a general strategy for forward pruning in chance nodes in a game like *Stratego*. We plan to combine this with an improvized alpha-beta pruning approach [5] in a way that will satisfy the rules of our game as well as have good performance. Both the previous techniques have only been tested on popular games and so they might perform poorly if implemented "as-is" in our game.

III. Approach

To reduce the complexity of the implementation and to fit into a workable time frame we made a conscious design decision to go with a 2 player, 5x5 board.

We formulated three different algorithms and pitted them against each other and calculated their winning rate in order to evaluate their effectiveness. The algorithms A1, A2 and A3 examine the current state of the board and based on the value of the dice, select the pawn to move by either:

Naive Agent A1: using a weighted tree approach that computes the naive evaluation function from the heuristics defined (Ref. to Algorithm 1 in Appendix E).

Intelligent Agent A2: using a modified version of the Expectiminimax [3] algorithm, that accounts for chance nodes by expanding all the Min nodes of the second player (= no. of dice values * no. of pawn values) and obtaining an average of all expanded values. This is a valid assignment as all the 5 dice values are equally probable (Ref. to Algorithm 2 in Appendix E).

Random Agent A3: selecting a random pawn through a normal (Gaussian) distribution.

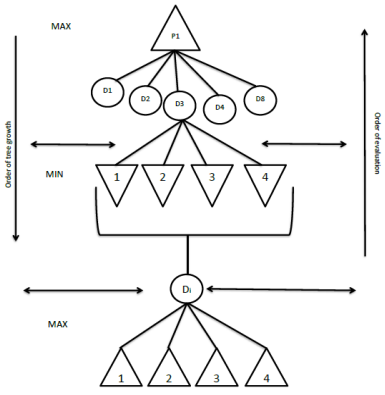

Fig. 2: Game Tree showing the expanded nodes for Pawn P1 (Steps 1-8 below)

Figure 2 is a diagrammatic representation of the working of A2.

1) We obtain the naive heuristic function of P1 based on the current configuration of the board ($N1$).
2) The chance nodes, $D1, D2, D3, D4, D8$ indicate the chance of Player 2 getting the dice values 1, 2, 3, 4, 8 respectively.
3) For each possible dice value for player 2, we calculate the naive heuristic function of each pawn assuming that player 1 has moved P1 (step 1). The figure shows it being done for $D3$.
4) For each of $D1, D2, D3, D4, D8$ player 2 will then have a pawn (P2) with the highest heuristic. Let this heuristic be $HMin$ = -1* $naiveEvaluation$ for player 2. At this level there will be 5 possible pawns (P2) of player 2 which can be moved.
5) For each P2 moved, player 1 will now calculate naively the heuristic for each of his pawns, for a given value of the dice.
6) For each dice value, there will be one pawn(P3) of player 1 with the highest heuristic. Let this be $HMax$. For every value of $HMin$ there will be 5 values of $HMax$ (i.e., there will be 5*5 = 25 values)
7) Compute $N1$+$HMin$+$HMax$ for each ordered pair $(HMin, HMax)$.
8) Since $D1, D2, D3, D4, D8$ are equally likely, find the average of the sum of all computed values from the previous step and update the evaluation of P1.
9) Repeat steps 1-8 for pawns P2, P3 and P4 and choose the pawn with the highest evaluation.

Heuristic Functions

Heuristic functions for evaluating nodes are constructed as a linear combination of the factors below, each with a particular weight.

F1: Will moving of this pawn hit an opponents pawn?
F2: Will the pawn move onto a safe house?
F3: Will the pawn be out-of-reach of an opponent?
F4: Will it result in the formation of a double?
F5: The closeness of the pawn in reaching the goal
F6: The closeness of a pawn in reaching the safesquare
F7: Number of remaining pawns on the board for the player

The heuristic functions formulated with F1-F7 are:

H1: $w1 * f1 + w2 * f2$,where $w1(= 10) > w2(= 8)$. This represents an attacking strategy where the first thing a player looks for is whether he is hitting an opponents pawn rather than looking out for his own safety

H2: $(w1 * f1 + w3 * f3) * f7/f5$,where $w3$ = *distance of the closest opponent pawn*. This takes into account the position of the pawn relative to the opponents pawns. The pawn not only tries to run away from the other pawns but this tendency increases as it reaches the goal square, depending on the number of the player's pawns remaining.

H3: $f1/f6$ This heuristic represents the tradeoff between an attacking and defensive strategy, where the player can decide on switching between the two on the fly.

H4: $(w1 * f1 + w2 * f2 + w3 * f3)/(w5 * f5 + w6 * f6)$ This holistic heuristic takes into account all the factors that we think are important, where $w1(= 10) > w2(= 8) > w3$. $w6$(=*number of squares from a safe square*)$< w5$(=*number of squares from the goal square*) account for the progress of the game with time.

We evaluated each of these heuristic functions and determined the best one to be used with A1 and A2.

We built the game in Python and pyGtk. For the purposes of the paper, we do not use the GUI in obtaining the results but it exists to serve as a future interface for a human player vs. computer player game.

IV. Results and Evaluation

We were interested in evaluating the following parameters:

1) *Efficiency of the algorithms used in determining the next move.* This is a comparative study of each algorithm against the other by measuring the number of wins across 50 games. In games involving A2 vs. A1, both of them use the same heuristic. (Ref. Tables I, II and III)

TABLE I: Intelligent Agent(A2) vs. Random Agent(A3)

Heuristic	Intelligent	Random	Average No. of Moves
H1	44	6	113
H2	40	10	107
H3	37	13	121
H4	47	3	111

TABLE II: Naive Agent(A1) vs. Random Agent(A3)

Heuristic	Naive	Random	Average No. of Moves
H1	43	7	116
H2	43	7	102
H3	31	19	142
H4	41	9	117

TABLE III: Intelligent Agent(A2) vs. Naive Agent(A1)

Heuristic	Intelligent	Naive	Average No. of Moves
H1	25	25	118
H2	22	28	100
H3	40	10	164
H4	27	23	117

2) *Efficiency of the heuristic used.* In order to evaluate this, we ran 50 games where A1 played against itself with each of the heuristics and the winning rates are recorded in Table IV. This was repeated for A2 as well (Table V).
3) *Branching factor of A2.* At any point in the game, a player has the ability to move at most 4 pawns. Since the dice can have 5 possible values, the branching factor for the game is 4*5 = 20.
4) In addition, for each of the algorithms, we were interested in computing the *time taken per move* and the *number of nodes expanded*. These are an average measure over 50 games and are listed in Tables VI and VII in Appendix D.

TABLE IV: Evaluation of each of the four heuristics for the algorithm A1

Player 1	Player 2	P1 won	P2 won	Average No. of Moves
H1	H2	27	23	116
H1	H3	42	8	163
H1	H4	33	17	116
H2	H3	36	14	126
H2	H4	31	19	112
H3	H4	16	34	138

TABLE V: Evaluation of each of the four heuristics for the algorithm A2

Player 1	Player 2	P1 won	P2 won	Average No. of Moves
H1	H2	28	22	139
H1	H3	33	17	170
H1	H4	32	18	146
H2	H3	26	24	120
H2	H4	23	27	114
H3	H4	27	23	126

V. Discussions

The most straightforward part of the evaluation was the time and space complexity analysis. Since A2 is a slight modification of EXPECTIMINIMAX, we expected it to take $\mathcal{O}(b^d)$ time and space, which is reflected in Tables VI and VII. The solution may not be obtained in the most optimized manner since we were unable to prune the game tree efficiently nor apply CHANCEPROBCUT on the game tree due to the rigid nature of the game rules.

As we can see from Tables I and II, we know that the Random Agent is trumped by the AI agent in most cases. We attribute the few wins of the Random Agent to chance. These two tables assure us that we can build an Intelligent Agent for the game of Chowka Bhara using an Adversarial Search Algorithm that factors the heuristics mentioned in the approach. They also give us a general idea about the heuristics; that H1, H2 and H4 are better than H3, but we will discuss the heuristics in greater detail when we get to Tables IV and V.

Table III, on the other hand, is a comparison between the two AI agents that we built. It aims to answer the question, *is a search algorithm better than a weighted tree approach[1]?* Not surprisingly, our initial assumption, that an adversarial search algorithm can be used to solve this game, holds true. What is interesting though is that A2, which looks down two levels to choose a pawn, does not always beat A1 which evaluates pawns based only on the current state of the board. In general, A2 performs better against A1, especially when they both use the heuristics H3 and H4. This shows that heuristics which combine both attacking and defensive strategies perform better when utilized for the agent looking down several levels as opposed to a purely defensive (H2) strategy. We can thus conclude that the efficiency of the search algorithm is largely dependent on the heuristic used.

Another question that arose was *will the Intelligent Agent be able to perform better if it is able to look deeper into the search tree?* This is a question that we will not be able to answer with precision until we are able to determine the most efficient heuristic for the game. Although, the results we have established so far can be extrapolated and we think that the performance of agent will be better.

We also observed that the average number of moves in games involving an AI agent (A1 or A2) and the random agent was around 110-120, whereas the number of moves in games between AI agents (A1 vs. A2, or A1 vs. A1 and A2 vs. A2 with different heuristics) was 130+. Interestingly, whenever the AI player won against the random player, the game ended in much fewer moves (90-100) compared to those (relatively sparse) games in which the random player won (190+). This goes to show that our AI agent always chooses the best possible move at each stage which reaffirms our assumption that the games won by Random Agent are by chance.

From Tables IV and V, we can conclude that for the weights used, H1 is the best heuristic for both the Naive (A1) and Intelligent agent (A2). If we consider only games with A1 vs. A1, the heuristics can be ranked as follows $H1 > H2 > H4 > H3$. However, for games with A2 vs. A2, apart from the clear victory of H1, we cannot clearly determine the relative ordering of H2, H4 and H3. What is counterintuitive however, is the low performance of H3. As a human player, one would definitely try to reach a safe square if possible, but it is interesting to see that the results tell us a different story.

Although it was interesting to build the game from scratch and come up with an AI player for it, we were unable to leverage existing algorithms in our favor nor build on an existing skeleton of the game. However we were successful in building a highly competent AI player for the game as well as unearth with certainty, heuristics that can be used to evaluate the game board. We learnt the importance of the weights used for each function in the heuristics.

We believe that by altering the weights, we can achieve a higher winning percentage for the Intelligent agent. In future, we need to construct an algorithm that learns the probabilities of winning against the various sequences of dice values generated and alter the weights of the heuristics based on its learning set, similar to a neural net or a genetic algorithm. It would also be interesting to extend our algorithm to facilitate a 4-player game and evaluate our heuristics for that scenario.

References

[1] Dhanan Sekhar Edathara (2012) Flash implementation of a variation of *Chowka Bhara* [Online]. Available: http://kavidikali.com/making/

[2] Russell, Stuart J.; Norvig, Peter (2010), Artificial Intelligence: A Modern Approach (3rd ed.), Upper Saddle River, New Jersey: Prentice Hall, pp. 165-167, ISBN 0-13-604259-7

[3] Russell, Stuart J.; Norvig, Peter (2010), Artificial Intelligence: A Modern Approach (3rd ed.), Upper Saddle River, New Jersey: Prentice Hall, pp. 177-180, ISBN 0-13-604259-7

[4] Schadd, M.P.D.; Winands, M.H.M.; Uiterwijk, J.W.H.M.; , "CHANCEPROBCUT: Forward pruning in chance nodes," Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on , vol., no., pp.178-185, 7-10 Sept. 2009.

[5] Zhang Congpin; Cui-Jinling; , "Improved Alpha-Beta Pruning of Heuristic Search in Game-Playing Tree," Computer Science and Information Engineering, 2009 WRI World Congress on , vol.2, no., pp.672-674, March 31 2009-April 2 2009

Appendix A - Historic Background of Chowka Bhara

Chowka bhara is a two- or four-player board game from India. This game is an example of a partially observable system that involves an element of chance introduced by the roll of a ”special” dice and an element of strategy (the strategy being the pawn the player decides to move after the roll of the dice).

The game of Chowka Bhara is one of the oldest board games, still being played in certain parts of India. There are references of this game in some ancient Indian epics like the Mahabharata. This game has been traditionally played on a silk cloth-lined board and with 4 cowry shells, which are the special dice.

Appendix B - Board Setup

Chowka bhara normally has a 5x5 square and four players, but one can also increase the number of squares depending on the number of players to any odd number x odd number (say 11x11). Assuming the size of the board is NxN (with N being odd), then each player will have N-1 pawns.

The 5x5 version looks as shown in the image below - there will be four players each having four pawns, starting at different positions (say North, East, South, West). They will start from the four crossed squares at the outermost ring and each player takes a turn to roll the shells. The mouth of the shell is considered to be of value 1 and the other end is of value 0. However, if every shell manages to show a value 0 (i.e land on the other end), then the value is considered to be 8. Therefore, the various values of this ”special” dice are 1, 2, 3, 4 and 8. If a player has cast the values 4 or 8, then he/she will get an additional turn. This may progress until that player gets a number other than 4 and 8 (i.e 1,2 or 3).

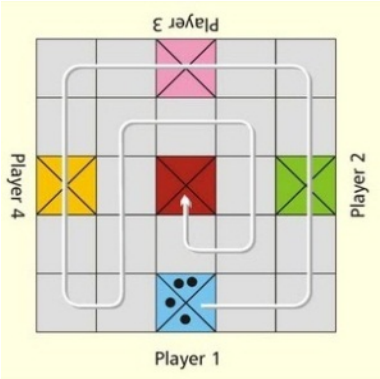

Fig. 3: A typical 5x5 board with the predefined path for player 1

Depending on the number the player gets, he/she can move one (or more) of their pawns that many number of squares on the board. Each player has a fixed path to move his/her pawns, which is in an anti-clockwise direction. The path for Player 1 is given in the figure above. Each player must completely traverse the outer squares, before stepping into the inner squares.

Appendix C- Rules of the game

Objective

For a player to win, he/she must move all their pawns to the center square.

Rules

This game has several intricate rules which need to be followed. Although there are several variations of this game, the following rules are for the standard implementation.

1) A player casts the shells to determine the number of square his/her pawns can move. If a player has cast 1,2 or 3, then he/she needs to choose one of their pawns and move it that many squares along the path designated for that player. The player needs to be able to smartly choose a pawn to move, so that it optimizes his chance of winning the game. If a player has cast either a 4 or an 8, the player will have another turn to cast the dice. The player does not have to move any of his pawns until he has cast 1,2 or 3.
   For example if a player casts a 4, he will get another chance to roll the dice. If on the second turn he gets a 3, then the player can move one of his pawns 4 squares and one of his other pawns 3 squares. He can of course choose to move the same pawn (4+3) 7 squares if he wants to. This argument can be extended to the player getting 3 or more consecutive turns.
2) **“Hit”:** Pawns of two different players cannot exist in the same square, other than a ”Safe” square, which are marked with an X in the figure. For a 5x5 board this is simply the starting positions of each of the players and the center square. However for higher dimension boards, more safe squares can be added symmetrically across the board.
   So if the pawn of player X lands on the same square of a pawn of player Y, then player X has ”hit” player Y. Player Y’s pawn is returned to its starting square and this pawn needs to start over. For a player’s pawn to progress into the inner squares, he should have ”hit” atleast one of his opponent’s pawns. This condition is imposed on the player and not on his pawn. For example, even if one of the player’s pawns has hit one of the opponent’s, then all his other pawns will be eligible to enter the inner square. If it so happens that a player cannot move any of his pawns because he has not ”hit” any of his opponents, then the player will lose that turn.
3) **“Double”:** It is possible for a player to have two of his pawns in the same square. This is called a ”double”. If a player forms a double on the outer square, then it blocks the opponent’s pawns behind him for one move; i.e if an opponent’s pawn crosses the double, then that move is voided and the opponent loses a turn. However on the next turn for the same opponent, his pawn can go past (or ”cross”) the double. This rule is applicable for every opponent of the player in the game. This rule is not valid once the double is formed on the inner squares. For the player forming the double, there are two choices

on his subsequent moves. He can "break" the double by moving only one of the pawns in it, or keep the double and advance the pawns together. In case a player has got multiple turns (as he may have thrown 4s and 8s), he can treat the double as one pawn and make it move to one of the values on the dice.

4) **Double vs Single:** It is not possible for a single pawn to "hit" a double. A player cannot move his single pawn to the same square as an opponent's double. A player cannot move past an opponent's double for 1 move. However, a double can "hit" a single pawn, and make it move back to its home square.
5) **Reaching the central square:** A pawn needs to reach the central square exactly. For example, if a pawn is 3 squares away from the center and the player throws a 4, then that pawn cannot be moved. If that is the only pawn left for the player to move (which may be because of a variety of reasons such as being blocked by double, or by virtue of being the last pawn left), the player will lose his turn.

APPENDIX D: TIME AND SPACE COMPLEXITY ANALYSIS

TABLE VI: Time taken per move for each algorithm

Algorithm	Time taken per move (in ms)
A1	9.3
A2	198.12
A3	0

TABLE VII: Number of nodes expanded for each algorithm

Algorithm	No. of nodes expanded
A1	4
A2	4*(4*5)*(4*5) = 1600
A3	0

APPENDIX E: PSEUDO CODE

Algorithm 1 Pseudo Code for **A1**

function GETPAWNNAIVELY(diceValue)
  for all $pawn : ListOfMovablePawns$ **do**
    ▷ Naive evaluation
    $evaluationFunction \leftarrow Hn(pawn, diceValue)$
  end for
  ▷ Pawn corresponding to the maximum evaluation according to the heuristic
  return pawn[MAX(Hn(ListOfMovablePawns))]
end function

Algorithm 2 Pseudo Code for **A2**

function GETPAWNINTELLIGENT(diceValue)
  for all $pawn1 : ListOfMovablePawns$ **do**
    ▷ Naive evaluation
    $evaluationFunction \leftarrow Hn(pawn1, diceValue)$
    ▷ Depth-Level 0
    $assumedBoard \leftarrow copy(currentBoard)$
    $assumedBoard.movePawn(player1Pawn)$
    ▷ Assume player1 moves pawn
    for all $possibleDiceValue : (1, 2, 3, 4, 8)$ **do**
      for all $pawn2 : opponentPawnList$ **do**
        $potEval \leftarrow Hn(pawn2, possibleDiceValue)$
      end for
      $bestPawnValue \leftarrow MIN(potEval)$
      ▷ Ordered pair (Dicei, pawni)
    end for
    ▷ Depth-Level 1
    for all $element : bestPawnValue$ **do**
      $assumedBoardPosition.$
      $movePawn(getPawn(element))$
      ▷ Assume player2 moves pawn
      for all $pDiceValue : (1, 2, 3, 4, 8)$ **do**
        for all $pawn3 : myPawnList$ **do**
          $secondEval \leftarrow Hn(pawn3, pDiceValue)$
          $bestPawnValue \leftarrow MAX(potEval)$
          ▷ Ordered pair(Pawni, (Dicej, pawnj), Pawnk)
          $evaluationList \leftarrow addToList$
          $(evaluationFunction$
          $-current(potentialEvaluation)$
          $+current(secondLevelEvaluation))$
          ▷ Flattened out chance nodes
          $H(pawn1) \leftarrow Avg(evaluationList)$
          ▷ Since every order of event is equally likely, the evaluation function of each pawn would be the average of all possible evaluationList elements
        end for
      end for
    end for
  end for
  ▷ Retracing back to the pawn at depth-level=0 that lead to the highest evaluation after 2 levels
  return pawn[MAX(Hn(ListOfMovablePawns))]
end function