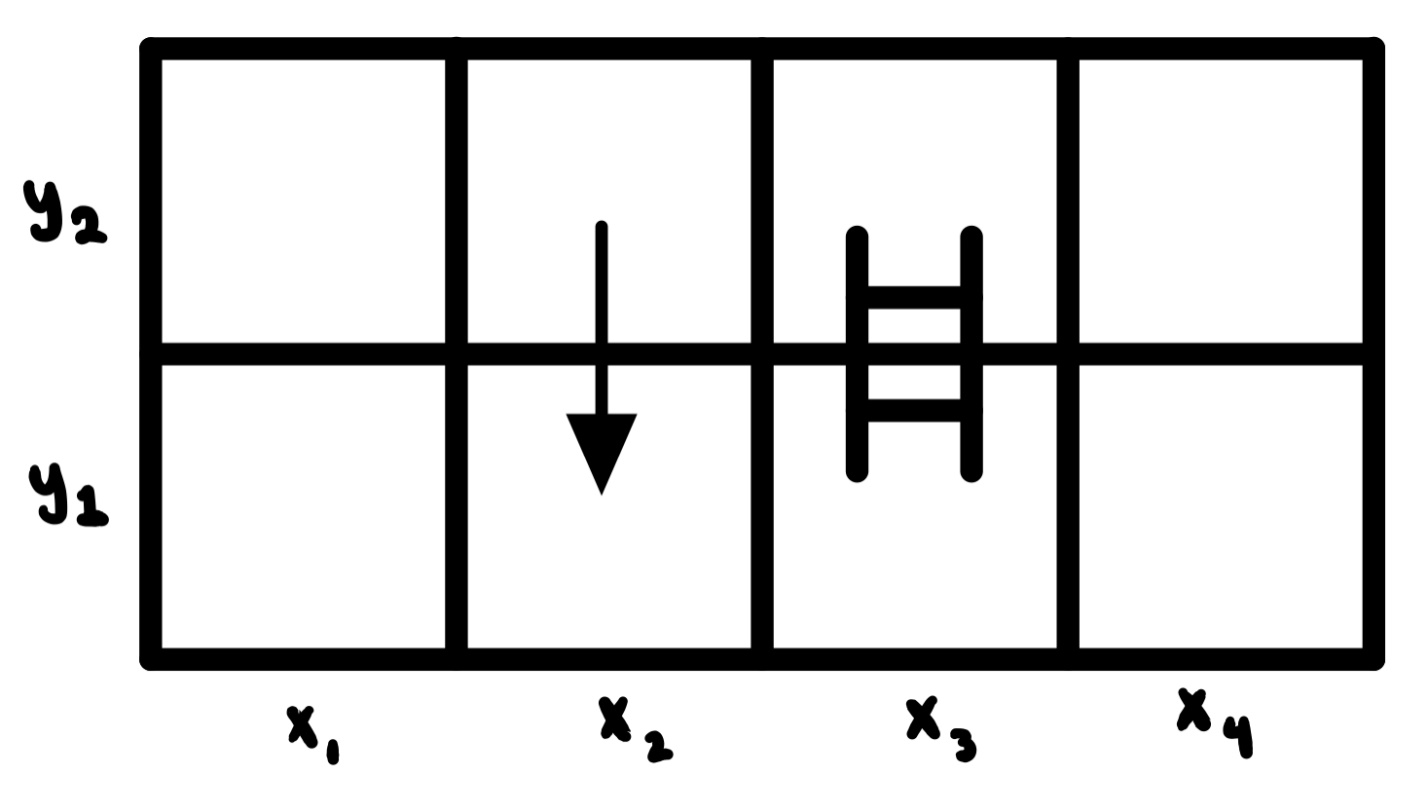
# Project Summary

Given a pre-set snakes-and-ladders board, a die with some number of sides, and a limited number of rolls, can a player starting at some starting position finish at the winning position, landing exactly on that winning position and use no more than the maximum number of rolls to arrive at that position? The size in both the x (horizontal) and y (vertical) directions of the board are pre-defined, where the length of the board in the x-direction is 4, and the length of board in the y-direction is 2. The number of sides on the dice are limited to 3, the number of players could be 1 or 2 depending on the user input, and the number of rolls for each player is limited to 3. An example snakes-and-ladders board is shown in Figure 1 below.



The starting position of one or both players is at the very bottom left of the board, where y=1 and x=1. The player(s) roll the dice and move horizontally in the increasing x-direction towards the bottom right of the board, y=1 and x=4. Then the player(s) move vertically from the bottom right towards the top right of the board, y=2 and x=4, before moving horizontally in the negative x-direction towards the top left of the board, y=2 and x=1. The winning tile on the board is the final tile, or x=1 and y=2 in Figure 1 above.

In addition to traditional movement along the board, there also exists a “snake” and a “ladder”. The “snake” pulls a player down in the negative y-direction, effectively setting the player back. The “ladder”, on the other hand, pushes a player up, moving the player closer to the final position faster. If a player were to land at the tail of the “snake” (the tail of the arrow in the figure above), the player would automatically shift down to the head of the “snake” (the head of the arrow in the figure above). If a player were to land at the bottom of the “ladder”, then the player would automatically move up to the top of the “ladder”. Since the top and bottom of the ladder look similar, know that ladders can only move a player in the positive y-direction. For example, if player 1 were to start at the proper starting position in Figure 1 above, and then roll a 3, the player would traverse the board horizontally towards y=1 and x=3, at which point the player will be automatically moved up to position x=3 and y=2.

The player(s) win only if they land exactly on the final tile in the board within the limit of the number of rolls, additionally constrained in movement by the number of sides on the dice. That means that if the player overshoots the board, that player loses. If the player hits the roll limit and fails to reach the final tile, that player also loses.

To implement this problem in logic, we created a variety of variables and arrays. The position of the player is determined by a combination of an array that stores Boolean values for x-positions, and a similar array that stores Boolean values for y-positions. A player at x=1 and y=1 would have True stored in the first element of both arrays. When the player moves from one position to the next, the original position’s x and y Boolean values stored in those 2 arrays are negated (made False), and the new x and y values are negated as well (made True). For example, if a player were to roll a 1 and move from the starting position to x=2 and y=1, the 1st position in the x-array is negated and becomes False, and the 2nd position in that same array is negated and becomes True. The x index and y index on the board correspond to the xth and yth elements in the arrays. We also created an array to represent the current turn of a player. The length of the array is the number of turns, where the current turn is made True, and all other turns are False. The 1st element in that array represents the 1st turn, the 2nd element the 2nd turn, and the 3rd element the 3rd turn. Using the arrays described above, the problem, including constraints and encodings can be implemented in logic.

To explore the model, we calculated the probability of a player wining, we added a 2nd player to the model to more closely simulate a real-life snakes-and-ladders game, and finally, throughout the implementation of the model, we made minor changes to the implementation to overcome errors in code.

# Propositions

**Preface**:“i” in the propositions below represents the y-position on the board, “j” represents the x-position on the board, and “k” represents the turn number. Additionally, the propositions described below are not implemented in code exactly the way they appear below. For example, the first proposition in the list of propositions below, suggests that one single variable carries the x and y-positions of the player, as well as the turn number. In the code, those 3 subscripted variables are separated into 3 distinct arrays, making implementation easier.

* Xi,j,k: True if the player is at position (i,j) at roll k where k = 1 represents the first turn
* Ry: True when y is rolled
* W: True if the player has won (i.e. landed perfectly on position X1,1 within the allotted number of turns
  + X1,2,k W: If a player lands on position X1,2 at turn k such that k is not greater than the maximum number of turns, then the player has won
* L: True if the player has lost (i.e. did not land perfectly at position X1,1 in the allotted number of turns
  + (X1,3,2 R3) (X1,3,1 R3) (X1,4,3) (X1,3,3) (X2,2,3)L: If the player rolls a 3 after 1 or 2 turns at position X1,3, or the player lands at position X1,4 or X1,3 or X2,2 after 3 turns, then the player has lost
* X2,2,k  X1,2,k: True when the player lands on position X2,2 (the tail of the snake) at turn k. The player is automatically moved to position X1,2 if the player lands on position X2,2
* X1,3,k X2,3,k: True when the player lands on position X1,3 (the bottom of the ladder) at turn k. The player is automatically moved to position X2,3 if the player lands on position X1,3
* Xi,j,k Ry X(i)(j+(-1)i\*y)(k+1) except X1,2 R3  X1,4,(k+1) and X1,4 Ry  X(i+1), (j-(y-1)), (k+1) : If the player rolls y at any position on the board with the exception of positions X2,2 and X2,4, use X(i)(j+(-1)i\*y) to determine the player’s landing position and increment the number of turns, k. This formula shifts the player to the right when i = 2 and shifts the player to the left when i = 1. For the case where the player is at position X1,2 and rolls a 3, the player lands at position X1,4 and the number of rolls, k, is incremented. For the case where the player is at position X1,4 and rolls y, use X(i+1,j-(y-1)) to determine the player’s landing position and the number of rolls, k, is incremented. This formula considers the vertical shift and the leftward shift when i goes from i = 2 to i = 1

# Constraints

* Model describes a single winning position where k is less than or equal to the maximum number of rolls
  + X2,1,k W
* Model limits the number of rolls to 3
  + Xi,j,k where k cannot be greater than 3
* Model constrains the dice to be 3-sided, with number 1, 2 and 3 on one of the 3 sides
  + Ry for y = 1, 2, 3
* Player starts at position X1,1
  + X1,1 always starts off true
* Player can only occupy one position at a time
  + Every time a player leaves a position, that position becomes false

# Model Exploration

1. **How likely it is that the player wins**

There are possible combinations of 1, 2, and 3 that can be rolled. The following table outlines whether each combinations results in a win or a loss:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1, 1, 1 | L | 2, 1, 1 | L | 3, 1, 1 | L |
| 1, 1, 2 | W | 2, 1, 2 | L | 3, 1, 2 | L |
| 1, 1, 3 | L | 2, 1, 3 | L | 3, 1, 3 | W |
| 1, 2, 1 | L | 2, 2, 1 | W | 3, 2, 1 | L |
| 1, 2, 2 | L | 2, 2, 2 | W | 3, 2, 2 | W |
| 1, 2, 3 | L | 2, 2, 3 | W | 3, 2, 3 | L |
| 1, 3, 1 | L | 2, 3, 1 | L | 3, 3, 1 | L |
| 1, 3, 2 | L | 2, 3, 2 | L | 3, 3, 2 | L |
| 1, 3, 3 | W | 2, 3, 3 | L | 3, 3, 3 | L |

Of the 27 possible combinations, 7 of them resulted in a win, yielding a win probability of:

This is helpful to know as it gives an idea of how difficult the game is and, should an easier or more difficult game be required, the dice size, number of turns, and board layout can be modified and the new win probability can be calculated. Exploring the model this way allows for analysis on how modifying specific variables will affect the difficulty of the game; for example, if the die size was reduced to 2, it may seem at first like this will make the game harder to win as smaller numbers on the dice mean each player will not move as far each turn, but this actually makes the game easier to win, with an increase to 37.5%, due to the nature of the snakes and ladders on the board.

We can also calculate the specific win percentages after a player’s first roll:

This shows us that a user’s probability of winning the game increases if their first roll is a 2 and decreases otherwise. This is helpful information to know in situations where you may want to see how a players likelihood of winning changes over time; for example, if this game were to be modified to be played in a casino setting, knowing how a player’s odds change throughout the course of the game would be very important for both the player and the casino.

1. **Overcoming issues during the process of code implementation**
   * For our code implementation to function as an actual game and not an instantaneous execution, we broke up our constraints into multiple encodings.
   * This not only served to make the program more entertaining during use, but also made debugging of individual constraints within each encoding easier to understand and solve.
   * When our program would behave unexpectedly, we would print the dictionary of values returned by our problem encoding with Encoding.solve(). This allowed us to see what a possible set of legal values assigned to our NNF Var’s could be, and then we would use that wrong assignment as a counterexample to track down the issue.
   * It's important to note that, where possible we tried to generalize our implementation, however the limitations of propositional logic in some cases made this challenging. An implementation of our first order extension would likely yield a much more general model.
2. **Extend the game to include 2 players**
   * We explored this model to make the game simulate a real-world snakes-and-ladders game more closely.
   * The logic used for a two-player game is very similar to a one player game, with each player having its own logic variables for their position.
   * The turn limit was removed when two players are playing the game. The game only ends when one player has reached the final square.
   * As a result, we correctly modeled a 2-player snakes-and-ladders game, where the game ends when one player reaches the final position and wins the game.

# Jape Proofs

1. Rolling a 2 will never result in a loss. (R2 implies not a loss). To simplify, only tiles on the second row (y = 2) are being considered as a roll of 2 will not be enough to win or lose from any tile on the first row. Also, tiles (x=1, y =2) and (x=2, y=2) are impossible as they represent the winning tile and the beginning of a snake, respectively; therefore, only two tiles need to be considered and this is represented in the propositions by .

**Propositions:**

(being on x3 and rolling a 2 implies a win)

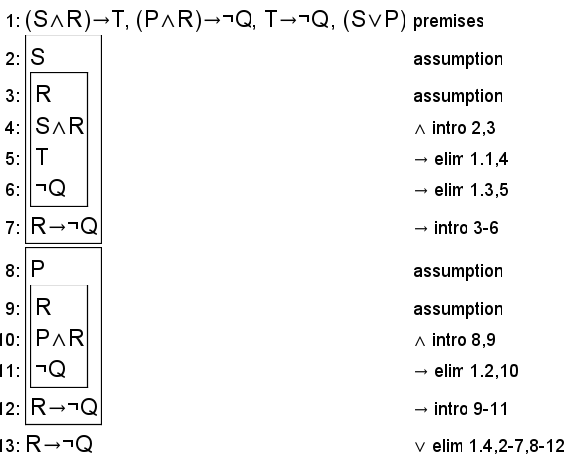
(being on x4 and rolling a 2 implies you haven’t lost)

(winning implies you have not lost)

(current position is either x3 or x4)

**Conclusion:**

(rolling a 2 means you have not lost)



**S represents x3**

**P represents x4**

**R represents r2**

**T represents W**

**Q represents L**

1. Rolling a 1 will never result in a win or a loss. The same simplifications have been made as the first proof.

**Propositions:**

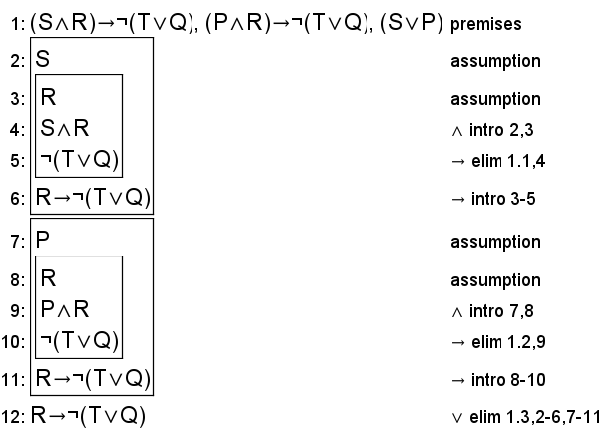
(being on x3 and rolling a 1 implies you haven’t won or lost)

(being on x4 and rolling a 1 implies you haven’t won or lost)

(current position is either x3 or x4)

**Conclusion:**

(rolling a 1 means you have not won or lost)



**S represents x3**

**P represents x4**

**R represents r1**

**T represents W**

**Q represents L**

1. You cannot win without at least two of your dice rolled being the same number. (ie. you cannot win if you roll 1, 2, and 3 each once in any order). In order to simplify, the given propositions for this sequent outline how the game state is affected after each of the 6 combinations of 1, 2, and 3 instead of outlining the game logic for each individual turn.

**Propositions:**

(rolling 1, 2, and 3 in those orders moves player to x­2,y1

(rolling 213 moves player to x4,y2)

(rolling 231 moves player past win square)

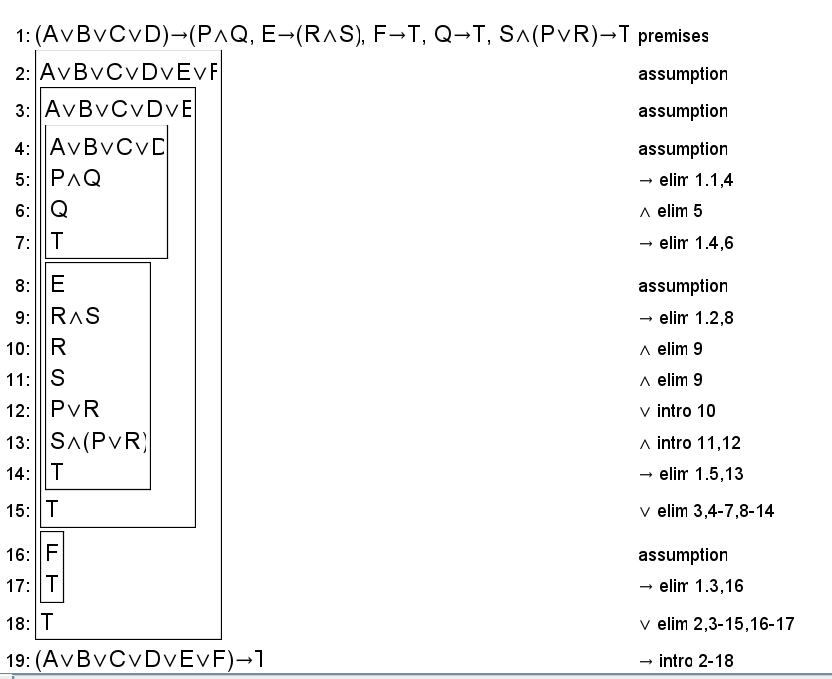
(ending on the bottom row means player hasn’t won)

(ending on second row but not win square means player hasn’t won)

Note – to simplify, the last proposition was represented as in Jape as x1, x2, x3, and x4 are mutually exclusive and x1 and x­­­­3 are not used elsewhere in the proof.

**Conclusion:**

(any combination of rolling 1, 2, and 3 means loss)



**A represents R123**

**B represents R132**

**C represents R312**

**D represents R321**

**E represents R213**

**F represents R231**

**P represents x2**

**Q represents y1**

**R represents x4**

**S represents y2**

**T represents L**

# First-Order Extension

In order to extend our model and incorporate predicate logic, the following predicates can be used:

B(x,y) to represent whether or not a square is a valid block (ie. The x,y coordinates are in the game’s domain)

,

This predicate can be used to make general statements applying to all blocks:

Ie. “for all blocks, rolling a 2 will not result in a loss (passing the win square)”

It can also be used to make statements pertaining to a specific row/column

Ie. “there exists a block in the first row that rolling a 1 takes you to the second row”

W(x, y, r) - represent if a roll r from block (x, y) will result in a won game

This predicate can be used to state it is possible to win in one roll from a given square:

It can also be used to state it is possible to win from any square with one given roll:

L(x, y) and S(x, y) to represent if the x, y coordinates represent a block that contains the beginning of a ladder or snake, respectively.