Reese Ingraham
ECON 305
Prof. Williams
11/15/18

Sushi Go Readme and Writeup

1 Introduction and How To Use This Software

This section will outline the components of this readme, as well as give a brief overview about how to use the Sushi Go simulation. This readme is broken into four parts: the introduction, which is the part you are currently reading, a write up about how my group's play session went, the design methodology, which will explain the design of the game in abstract as well as translate the abstractions into Java code, and finally an analysis of the code and simulations, and suggestions for future improvements.

In order to use the software, just run the Java code in any IDE or vim and the console output will tell you what is happening in the game. It is important, however, that the *Game.java* and the *Player.java* files are in the same folder, otherwise the *Game* class can't read in the *Player* class.

However, there are a multitude of errors, specifically one hundred. I think that there are more errors, but jGrasp gives up at 100. Specifically, the errors seem to revolve around the *dealCards* method, in which it says it expects greater than and less than signs, square brackets, semicolons, and other symbols in the conditional, as well as saying that the conditional isn't a statement. Technically, it's correct in saying that the conditional isn't a statement, but it's not attempting to be. If I had more time to debug it, I'm sure I could figure it out, but I've already sunk around sixteen arounds into this. It also doesn't help that this is my first time programming in Java in about eight months.

2 Playthrough Writeup

Reese Ingraham
ECON 305
Prof. Williams
11/15/18

I used a simplified strategy that I gleaned from looking at the rules, in addition to the Sushi Go subreddit. As Felix kindly described it in his presentation, it was the "least stable" strategy of the four strategies, as I tended to hoard all of the wasabi at the very beginning, which lead to a very high-risk-high-reward sort of playstyle. I do typically enjoy that when I play games, but it probably lead to a slightly lower expected payoff than I would have liked. I think that Harrison's formal strategy was the most complete and the most efficient. People tended to win with Harrison's strategy most of the time. Felix's was also very good, while Chengdong's was a little hard to follow, especially with a fast-paced game like Sushi Go.

However, once we abandoned our formal strategies I think Chengdong and I fared the best. I can't quite articulate what the difference was. I think perhaps my formal strategy was along the right lines, as I believe I still played fairly similarly to the formal strategy, but I think abandoning the rigidity of the strategy allowed me flexibility to change my strategy on the fly, while still prioritizing all the things I prioritized in my formal strategy.

In addition, unlike my formal strategy, I didn't use mixed strategies in my normal play session. This is because Sushi Go, unlike a game like Rock Paper Scissors, does not lend itself to using mixed strategies. You're not really trying to trick your opponents, and in most cases there's probably some corner solution in terms of the cards you can/should play. My formal strategy was also very aggressive. After abandoning the formal strategy, I tended to play more defensively, which I think is a key component to being an effective Sushi Go player. I think that a good mix of offensive and defensive playing is key to maximizing the points differential between yourself and other players.

3 Design Methodology

Reese Ingraham
ECON 305
Prof. Williams
11/15/18

3.1 Design Methodology Introduction

This section will serve to provide insights as to why I made the choices that I made for this game. This also serves as a notebook of sorts for me to keep my thoughts organized while I'm designing the game. This section serves as the introduction, whereas the next section .

3.2 The Players

In our simulation of Sushi Go, we only have two players, let's call the players *i* and *j*. In Java, I wrote a custom player class to store all of the important information that we want our program to keep track of. There are five essential components of a player that we want our program to keep track of:

1. Name, so we can see which player is which, which is kind of important.
2. Hand. The player's hand will dictate what actions a player *can* take.
3. Table Cards. These will inform what our player's score is.
4. Score. Some might say this is also important for analyzing a game, but who knows.

The players will be named *a* and *b*. The game is symmetric, so everytime I talk about player *a*, it will also apply to player *b*, unless otherwise noted. The name is stored as an integer called name. The actions are the cards the player can play, and which will be noted by the set $\sigma_i = \{a_i^1, a_i^2, \ldots, a_i^n\}$, where a_a^t denotes an individual action player *a* can take, i.e., a card that the player can play in period *t*. The actions are stored in a one-dimensional array of integers called actionArray.

3.3 The Actions

As mentioned in 3.2.1, the actions available to a player are the cards they currently have in their hand, which is represented by the hand array. Notice that I didn't mention any conception

of strategies in the player section of the document. This is because the players don't have any strategy that's unique to themselves; they share a strategy. Essentially, player *a* attempts to maximize total utility Π_i, which is given by $\Pi_i = \sum_{t=1}^{T} \pi_a^t$, where π_a^t is the payoff to player *a* in period *a* as a function of a_a^t and a_a^t, and *T* is the total number of periods. I made the assumption that player *a* can maximize Π_a by attempting to maximize each individual π_a^t. There is no requirement that this be a dominant strategy, but it makes the math and coding easier, as well as the fact that it'll get us pretty close. The only reason that it may not be a dominant strategy is that it may be advantageous for player *a* to play a card in period *t* that doesn't maximize π_a^t, but it prevents player *a* from making certain actions, in addition to cards that don't grant any points in the current period, but do modify the points in the future periods, like Wasabi. Thus, because player *a* will hedonically maximize π_a^t in order to maximize Π_i, we are thus creating a greedy algorithm.

The available actions are determined by what cards each player is dealt. The card dealing algorithm is given as follows. Let $deck[l]$ represent the *l*th entry in the *deck* array, and *r* is a uniformly distributed random variable on the integers from 1 to $\sum_{l=0}^{11} deck[l]$, inclusive. If it is player *l*'s turn to receive a card, then which card they receive depends on the instantaneous value of *r*, as follows:

Case 1: $0 \leq r \leq deck[0] \rightarrow$ player *i* receives a Tempura.

Case 2: $deck[0] + 1 \leq r \leq \sum_{l=0}^{1} deck[l] \rightarrow$ player *i* receives a Sashimi.

Case 3: $\sum_{l=0}^{1} deck[l] + 1 \leq r \leq \sum_{l=0}^{2} deck[l] \rightarrow$ player i receives a Dumpling.

Case 4: $\sum_{l=0}^{2} deck[l] + 1 \leq r \leq \sum_{l=0}^{3} deck[l] \rightarrow$ player i receives a 2 Sushi Roll.

Case 5: $\sum_{l=0}^{3} deck[l] + 1 \leq r \leq \sum_{l=0}^{4} deck[l] \rightarrow$ player i receives a 3 Sushi Roll.

Case 6: $\sum_{l=0}^{4} deck[l] + 1 \leq r \leq \sum_{l=0}^{5} deck[l] \rightarrow$ player i receives a 1 Sushi Roll.

Case 7: $\sum_{l=0}^{5} deck[l] + 1 \leq r \leq \sum_{l=0}^{6} deck[l] \rightarrow$ player i receives a Salmon Nigiri.

Case 8: $\sum_{l=0}^{6} deck[l] + 1 \leq r \leq \sum_{l=0}^{7} deck[l] \rightarrow$ player i receives a Squid Nigiri.

Case 9: $\sum_{l=0}^{7} deck[l] + 1 \leq r \leq \sum_{l=0}^{8} deck[l] \rightarrow$ player i receives an Egg Nigiri.

Case 10: $\sum_{l=0}^{8} deck[l] + 1 \leq r \leq \sum_{l=0}^{9} deck[l] \rightarrow$ player i receives a Pudding.

Case 11: $\sum_{l=0}^{9} deck[l] + 1 \leq r \leq \sum_{l=0}^{10} deck[l] \rightarrow$ player i receives a Wasabi.

Case 12: $\sum_{l=0}^{10} deck[l] + 1 \leq r \leq \sum_{l=0}^{11} deck[l] \rightarrow$ player i receives a Chopsticks.

3.4 Payoffs

The payoffs in Sushi Go are the points distributed to each player at the end of each round. The score is some intermediate value function π_i^t s.t. $\pi_i^t : a_i^t \times a_j^t \rightarrow \mathbb{Z}$. Let *table* be a two-dimensional array of length 10. This is denoted by $table[m][n]$, where m represents which which card is being referenced per the cheat-sheet, and n represents which player is being

Reese Ingraham
ECON 305
Prof. Williams
11/15/18

referenced, with 0 corresponding to player *i* and 1 corresponding to player *j*. Specifically, π_i^t is a piecewise function given by:

Case 1: a_a^l = Tempura

Subcase 1: $\pi_a^l = 0$ if *table*[0][0] % 2 == 0.

Subcase 2: $\pi_a^l = 5$ if *table*[0][0] % 2 != 0.

Case 2: a_a^l = Sashimi

Subcase 1: $\pi_a^l = 0$ if *table*[1][0] + 1 % 3 $\neq$ 0.

Subcase 2: $\pi_a^l = 10$ if *table*[1][0] + 1 % 3 = 0.

Case 3: a_a^l = Dumplings

Subcase 1: $\pi_a^l = 1$ if *table*[2][0] = 0.

Subcase 2: $\pi_a^l = 3$ if *table*[2][0] = 1.

Subcase 3: $\pi_i^l = 6$ if *table*[2][0] = 2.

Subcase 4: $\pi_i^l = 10$ if *table*[2][0] = 3.

Subcase 5: $\pi_i^l = 15$ if *table*[2][0] $\geq$ 4.

Case 4: a_i^l = any sushi roll.

Subcase 1: $\pi_i^l = 6$ if *table*[3][0] > *table*[3][1]

Subcase 2: $\pi_i^l = 3$ if *table*[3][0] $\leq$ *table*[3][1]

Case 5: a_i^l = Salmon nigiri.

Subcase 1: $\pi_i^l = 2$ if *table*[4][0] + *table*[8][0] % 2 != 0.

Subcase 2: $\pi_i^l = 6$ if *table*[4][0] + *table*[8][0] % 2 == 0.

Case 6: a_i^l = Squid Nigiri

Subcase 1: $\pi_i^l = 3$ if $table[5][0] + table[8][0] \% 2 != 0$.

Subcase 2: $\pi_i^l = 9$ if $table[5][0] + table[8][0] \% 2 == 0$.

Case 7: a_i^l = Egg Nigiri

Subcase 1: $\pi_i^l = 1$ if $table[6][0] + table[8][0] \% 2 != 0$.

Subcase 2: $\pi_i^l = 3$ if $table[6][0] + table[8][0] \% 2 == 0$.

Case 8: a_i^l = Pudding

Subcase 1: $\pi_i^l = 6$ if $table[7][0] > table[7][1]$

Subcase 2: $\pi_i^l = 3$ if $table[7][0] == table[7][1]$

Subcase 3: $\pi_i^l = -6$ if $table[7][1] < table[7][1]$

Case 9: a_i^l = Wasabi

Subcase 1: $\pi_i^l = 0$.

Case 10: a_i^l = Chopsticks

Subcase 1: $\pi_i^l = 0$.

3.5 Strategies

Technically, this should go under the Players heading, but it didn't make sense to talk about strategies before talking about the payoffs. Like I mentioned before, the Players strategy, initialized in the *choice* method, can be thought of as a greedy algorithm like Dijkstra's, where it maximizes immediate utility, which could be suboptimal, instead of planning for the future. This is why both Chopsticks and Wasabi, while valuable cards, give an intermediate utility of zero. I implemented this by creating a two-dimensional array of length 11 - what round we're on called

strategySet, where the columns represent what card we're talking about, the 0th (or first, however you want to think about it) row represents how many of that card is in the deck, and the 1st (or 2nd) row represents the intermediate valuation of that card given the current conditions. Then, I did a bubble sort on the two dimensional array such that the card in index 0 has the least intermediate valuation and the card in index 11 - round has the highest intermediate valuation. The choice is that the player picks is thus the card in the last index. This is probably not the most efficient way to perform this calculation, especially as bubble-sort has an asymptotic time complexity of $O(n^2)$, but it was a quick-and-easy method. In the main method of the program, *a*'s score and b's score is thus computed, and using a *for* loop, the winner is declared after 10 rounds.

3.6 Other Methods

A couple other quick methods I want to mention are the *switchHands* method, the *tableUpdate* method, and the *scoreUpdate* method. These are all really simple algorithms that I did not feel deserved their own section, but I felt that I should still talk about. In Sushi Go, you switch hands with your opponents after you play a card. In a two-player game of Sushi Go, this is very easy. I made a deep copy of one of the hands, replaced all the values of that hand with the other, and then replaced all of the values of the hand that I didn't make a deep copy of with the values of the deep copy.

The *tableUpdate* method is really simple; it inputs a card and a player, and it updates the player's table card accordingly.

Reese Ingraham
ECON 305
Prof. Williams
11/15/18

The *scoreUpdate* method is also very simple. At the end of the game, this method is called, and it iterates through all of the table arrays of both players, and calculates each player's score. The player with the higher amount of points is thus considered the winner.