David Amin (dma3fq), Rupali Vohra (rv5rr)

CS 4710: Homework 3

Negotiating Agents

Agent 1: rv5rr.py

Goal: For this agent, we set out with the intention to determine the opponent's optimal ordering. With this knowledge, we hoped to find the perfect compromise between the two agents, so that both would gain some (hopefully equivalent) utility.

Approach: This agent starts by figuring out all of the different possible permutations of the given preferences list. It also figures out the different utility[1] each permutation provides to itself. As we were programming, we realized that the utility was a function of the Hamming distance of an agent's perfect ordering to the offer it received. As such, there were a fairly limited number of levels of utility, which will be referred to as "utility buckets" in the rest of this description. We created a dictionary that mapped each utility bucket to a list of offers that would give the agent that utility.

Next came the challenge of figuring out what the opponent's preference was, and using that to find a compromise. We decided to create a weighted average of each object's location in each offer, with offers that gave the opponent higher utility weighted to have more of an effect in calculating the object's real location. After weighting these averages, we set the value with the lowest position calculation as the first element, the next lowest as the second element, and so on until we had an ordering that could potentially be the opponent's preference. We then passed this ordering as an offer to a function (find_highest) to find out what our best compromise was.

The find_highest function would cycle through each utility bucket and search for an offer that had the smallest Hamming distance between what we had calculated to be the opponent's preference and each potential offer in the utility buckets. In order to make sure we were not losing out and giving the opponent exactly what they wanted, the find_highest function also took in another parameter called bucket_num. This parameter limited how many of the utility buckets to go through, and the search always started with our agent's highest utility bucket. This bucket_num was adjusted based on the opponent's actions. If the opponent decided to make an offer that gave them higher utility than a previous offer, we would decrease bucket_num[2]. This meant we would search through one less lower utility bucket, meaning we were less compromising[3]. Similarly, if the opponent gave us a successive offer

---

[1] Since the utilities were given as a floating point number, and since it is not possible for floating point numbers to represent all numbers, we knew that we would get some numbers of identical utility that would be represented differently. To account for this, we decided to round off all floating point numbers at 5 decimal places, hoping to reduce the effects inherent in floating point arithmetic.

[2] It is possible to get negative buckets here. This can occur if the opponent makes successive offers that increase their own utility. If this happens, the function only searches through its highest bucket (which yields this agent's preference). The consequence to the opponent, however, is that they must make up the difference by offering successively lower utility offers so that our agent is more compromising by searching through more buckets.

[3] At any point, it was possible that by going to a different bucket, we would find a perfect match for what we thought was their preference. To prevent the opponent from getting more utility than we were getting by searching in a lower bucket, we would only ever return, at best, an offer that had two items switched for the opponent. The only exception to this rule was if they offered something that was our best preference.

that gave them the same utility as their last offer, we would not change the number of utility buckets we searched through. Lastly, if the opponent was compromising and gave us an offer that left them with less utility than before, we would increase the number of buckets we searched through.

Results: We ran our tests against a few other agents we had that were in their beginning stages. These were agents we had either started making and decided not to use, or snapshots of early phases of our working agents. One of the agents was the provided random agent. To test, we ran our agent against the same opponent for 3 consecutive negotiations with the same scenario for each negotiation. This happened twice for each scenario; once, the agent was negotiator A for three negotiations, and then the agent was negotiator B for three negotiations. The results of these tests are listed below, with the table showing our agents utility (rounded to two digits after the decimal), followed by a slash and then the opponent's utility (rounded to two digits after the decimal point).

Against the Provided Random Agent			
As A	Input size 5	Input size 7	input size 9
Round 1	**11.42**/3.42	-7/-7	-9/-9
Round 2	**9.42**/3.42	**16.15**/-1.85	**23.46**/9.46
Round 3	-5/-5	**16.15**/-1.85	-9/-9
Final	**15.83**/1.83	**25.30**/-10.7	**5.46**/-8.54
As B	Input size 5	Input size 7	input size 9
Round 1	**7.42**/5.42	**14.15**/-3.85	**23.46**/7.46
Round 2	**7.42**/3.42	-7/-7	-9/-9
Round 3	-5/-5	-7/-7	**25.46**/9.46
Final	**9.83**/3.83	**0.15**/-17.85	**39.92**/7.92

Against a negotiator that accepts offers that meet a certain threshold for utility. This threshold starts at the maximum utility and decreases at every round.

NegotiatorSimple			
As A	Input size 5	Input size 7	input size 9
Round 1	**11.42**/3.42	-7/-7	**25.46**/9.46
Round 2	**11.42**/3.42	**16.15**/-1.85	**25.46**/9.46
Round 3	**11.42**/3.42	**16.15**/-1.85	**25.46**/9.46
Final	**34.25**/10.25	**25.30**/-10.7	**76.38**/28.38
As B	Input size 5	Input size 7	input size 9
Round 1	3.42/**7.42**	6.15/**8.15**	**19.46**/15.46
Round 2	3.42/**7.42**	**10.15**/4.15	**17.46**/14.46
Round 3	3.42/**7.42**	6.15/**8.15**	**15.46**/14.46
Final	10.25/**22.25**	**22.45**/20.45	**52.38**/44.38

Against a negotiator that mimics its opponent's relative movement in terms of utility.

NegotiatorProb			
As A	Input size 5	Input size 7	input size 9
Round 1	**9.42**/5.42	-7/-7	-9/-9
Round 2	**9.42**/5.42	-7/-7	**23.46**/7.46
Round 3	**9.42**/5.42	-7/-7	**23.46**/7.46
Final	**28.25**/16.25	-21/-21	**37.92**/5.92
As B	Input size 5	Input size 7	input size 9
Round 1	3.42/**11.42**	6.15/**8.15**	-9/-9
Round 2	3.42/**11.42**	4.15/**10.15**	-9/-9
Round 3	1.42/**9.42**	4.15/**10.15**	-9/-9
Final	8.25/**32.25**	14.45/**28.45**	-27/-27

Against itself. Note that in the following table, the entries describe the utility of the Agent 1 specification (e.g. “As A”, “As B”), followed by a slash, followed by the other Agent 1. Since multiple runs yielded the same results, only one instance of Agent 1 is shown; Agent 1 “As B”’s results can be seen after the slash.

Agent 1			
As A	Input size 5	Input size 7	input size 9
Round 1	-5/-5	-7/-7	-9/-9
Round 2	-5/-5	-7/-7	-9/-9
Round 3	-5/-5	-7/-7	-9/-9
Final	-15/-15	-21/-21	-27/-27

Against the second agent, described later in this write up. Note that in the following table, the entries describe the utility of Agent 1, followed by a slash, followed by the utility of Agent 2.

Agent 2			
As A	Input size 5	Input size 7	input size 9
Round 1	-5/-5	-7/-7	-9/-9
Round 2	**9.42**/5.42	-7/-7	-9/-9
Round 3	**9.42**/5.42	-7/-7	-9/-9
Final	**13.83**/5.83	-21/-21	-27/-27
As B	Input size 5	Input size 7	input size 9
Round 1	5.42/**7.42**	-7/-7	**19.46**/13.46
Round 2	**9.42**/3.42	-7/-7	**19.46**/13.46
Round 3	**11.42**/3.42	-7/-7	**19.46**/13.46
Final	**26.25**/14.25	-21/-21	**58.38**/40.38

Analysis of Results: It is important to note that variation in results across rounds is to be expected. There is a bit of randomness built into this agent within the find_highest function’s looping mechanism, as well as potentially in the opposing agent. Also, if agents are learning as they progress, different results would be output.

Agent 1 either won or failed to reach an agreement in every match against the random agent. This is unsurprising, as the random agent is, as the name suggests, purely random. There is no strategy

to its progress. The failure to reach an agreement probably stems from the fact that Agent 1 was unable to pin down exactly what the random agent's preference was, and thus kept making random offers. There was only a 5% chance of an offer being accepted by the random function at every iteration, but that just did not end up happening.

Some interesting behavior came out when Agent 1 was pitted against NegotiatorSimple. When Agent 1 went first, it had a tendency to win (with one instance failing to reach an agreement), it had a tendency to win, but when it went second, it gave NegotiatorSimple the upper hand many times, especially in the instances with a smaller number of elements in the preference list. Since Agent 1 is very accommodating to "nice" agents (i.e. those you are willing to negotiate with lowered utility to themselves), and since NegotiatorSimple decreases its utility to itself at every iteration, it ended up unintentionally exploiting Agent 1. Although Agent 1 attempts to safeguard against giving the opposing agent its preference (details mentioned in the description of the agent), the weighted average estimation technique it uses to figure out that preference is not 100% accurate, which could have led to this loss. Also, since NegotiatorSimple lowers its utility every time, it causes Agent 1 to increase the number of buckets it looks through at every iteration, the ramifications of which have been discussed above.

Agent 1 had some interesting behavior against NegotiatorProb as well. Although Agent 1 won in most instances when it went first (when it did not fail to reach an agreement, that is), it had a tendency to end up with lower utility than its opposition. Since NegotiatorProb mimics its opponent's behavior, whenever Agent 1 went down in its utility, NegotiatorProb would have done the same thing. This process probably created an effect similar to the one that NegotiatorSimple created where Agent 1 searched through more buckets to find a compromising offer, and thus was too compromising.

When pitted against itself, Agent 1 always failed to reach an agreement. We think this is because both of the agents were too compromising, and since each one was trying to figure out the other's preference, by probing with other offers, neither was successful in doing so.

Against the second agent discussed in this paper, Agent 1 either won or failed to reach an agreement in most cases, regardless of whether it went first or second. This is probably because Agent 2 was attempting to predict the kinds of offers Agent 1 would provide, but Agent 1 was attempting to be somewhat compromising, so Agent 2 may not have been able to predict Agent 1's actions successfully.

Future Improvements: Currently, this agent does not learn any more through multiple rounds than it does from a single round. Since we find out the utility of an accepted offer, this agent could use that to its advantage in finding out the opposing agent's true preference ordering, along with combining all previous knowledge of the opponent's prior offers since their preference will not change across rounds.

Another potential improvement could be made in the permutations calculator. Though only called once per initialization, the time to compute every permutation is very long, compromising the agent's capability in runs with more elements (discussed further below). Creating a custom permutation finder that jumps around in the created permutations' Hamming distance to gain a sample for each utility bucket could save the agent a lot of time without compromising the intelligence of the system. This process would also use less memory.

Lastly, through testing we realized that Agent 1 was vulnerable to agents who were very giving at first. This means that an agent could make offers of lower and lower utility to themselves, causing us to increase our bucket size, and then dramatically increase their utility at the end. This would cause the agent to search only one fewer bucket, which could be meaningless if, through enough rounds and with a large enough number of utility buckets, the opposing agent had already increased our bucket search size significantly. A potential safeguard to prevent this from occurring would be to decrease the number of buckets the agent searches through proportionally to how drastic the change in their utility is for their given offer.

Agent 2: dma3fq.py

Approach: For our second agent, we decided to develop an incrementally more elaborate system. The benefit here was twofold, we were easily able to expand from the different points of our development as ideas struck, and we had a series of working agents to test against, to see if our changes were helping us improve relative to them. Initially, we built an agent that was quite simple; all it did was enumerate all possible permutations of the list, then categorized them by how much utility each one was worth, and then determined how many different "levels" of utility there were (by rounding the different utility values, which often tended to be the same based on how utility is calculated). We consider most of the problem from the perspective of moving "up" or "down" in these levels, which is roughly the same as making an offer with more or less utility respectively. To start, the agent had a minimum utility it needed to achieve. This minimum decreased as the rounds went on, and if at any point the agent received an offer of greater utility than the one it was currently on, it accepted that offer. Obviously, this is not very complicated. Another system could easily beat this agent by holding out and making offers that are bad for the agent until the very end, then making a single offer that is reasonable for our agent and great for theirs.

To improve on this, we decided to add a simple predictive scheme to the agent. The agent kept track of what the opponent had done in their moves so far, how many times they increased their own utility, how many times they conceded utility, and how many times they did not change. The agent used this information to determine the likelihood of the opponent taking a loss to utility, which was assumed to be a sign of cooperation. Strictly speaking, this assumption may not always hold true, but it is assumed for the purposes of the agent that any decrease in utility is a sign of good faith. If the opponent was likely to cooperate, our agent cooperated by taking a hit to utility in the offer it made to the opposing agent. If the opposing agent was likely to drive its utility higher, ours matched that by sending a deal with a higher utility for us. The benefit to this approach was that it would not fall victim to the mistakes of the first approach; it would tend to be tough with tough opponents and cooperative with cooperative opponents.

Our final approach, what we actually submitted as Agent 2 in dma3fq.py, is a slight but important refinement on the probabilistic technique of its predecessor. Rather than constantly moving in the direction the opponent is most likely to move, it moves randomly up, down, or not at all, with the same likelihood that the opponent would impact their own utility the same way. In this way, it doesn't fall into lowering its minimum utility with a cooperating opponent, then continuing to do so while the opponent is uncharacteristically uncooperative. The agent might move up or down, or even stay at the same amount of utility, even if an opponent cooperates or stonewalls negotiations, which allows for

some variability, and makes our agent more resistant to changes in the behavior of the opponent. This is additionally beneficial because it makes it harder for an opponent to read the strategy of the agent and exploit it.

Results: We ran our agent against a few others that we had available to see how it stood up to their strategies. To test, we ran our agent against the same opponent for 3 consecutive negotiations with the same scenario for each negotiation. This happened twice for each scenario; once, the agent was negotiator A for three negotiations, and then the agent was negotiator B for three negotiations. The results of these tests are listed below, with the table showing our agents utility, followed by a slash and then the opponent's utility.

Against the Provided Random Agent			
As A	Input size 5	Input size 7	input size 9
Round 1	**7.42**/-0.58	**18.15**/-3.85	-9/-9
Round 2	**7.42**/3.42	-7/-7	**23.46**/7.46
Round 3	5.42/5.42	**16.15**/-3.85	**17.46**/13.46
Final	**20.25**/8.25	**27.30**/-14.70	**31.92**/11.92
As B	Input size 5	Input size 7	input size 9
Round 1	5.42/5.42	-7/-7	**23.46**/9.46
Round 2	**11.42**/3.42	**16.15**/-3.85	-9/-9
Round 3	-5/-5	**16.15**/-3.85	**23.46**/7.46
Final	**13.83**/3.83	**25.30**/-14.70	**37.92**/7.92

Against a negotiator that accepts offers that meet a certain threshold for utility. This threshold starts at the maximum utility and decreases at every round.

NegotiatorSimple			
As A	Input size 5	Input size 7	input size 9
Round 1	**9.42**/5.42	**8.15**/0.15	-9/-9
Round 2	3.42/**5.42**	4.15/4.15	-9/-9
Round 3	**7.42**/5.42	2.15/2.15	**15.46**/11.46
Final	**20.25**/16.25	**14.45**/6.45	**-2.54**/-6.54
As B	Input size 5	Input size 7	input size 9
Round 1	**7.42**/3.42	**8.15**/2.15	-9/-9
Round 2	**9.42**/3.42	**8.15**/-1.85	-9/-9
Round 3	**7.42**/1.42	**8.15**/-1.85	**9.46**/-6.54
Final	**24.25**/8.25	**18.45**/4.45	**-8.54**/-24.54

Against a negotiator that mimics its opponent's relative movement in terms of utility.

NegotiatorProb			
As A	Input size 5	Input size 7	input size 9
Round 1	**9.41**/5.41	**6.15**/4.15	-9/-9
Round 2	5.41/**7.41**	**10.15**/2.15	11.46/**13.46**
Round 3	5.41/**7.41**	**8.15**/2.15	11.46/**13.46**
Final	20.25/20.25	**24.45**/8.45	13.92/**17.92**
As B	Input size 5	Input size 7	input size 9
Round 1	3.41/**7.41**	**10.15**/-1.85	9.46/9.46
Round 2	7.41/7.41	**8.15**/0.15	**11.46**/9.46
Round 3	**5.41**/3.41	2.15/**4.15**	13.46/**15.46**
Final	16.25/**18.25**	**20.45**/2.45	34.38/34.38

Against itself. Note that in the following table, the entries describe the utility of the Agent 2 specification (i.e. "As A"), followed by a slash, followed by the other Agent 2. Only one instance of Agent 2 is shown; Agent 2 "As B"'s results can be seen after the slash.

Agent 2			
As A	Input size 5	Input size 7	input size 9
Round 1	**11.42**/3.42	-7/-7	-9/-9
Round 2	**11.42**/3.42	**8.15**/2.15	-9/-9
Round 3	**9.42**/3.42	**8.15**/2.15	-9/-9
Final	**32.25**/10.25	**9.30**/-2.70	-27/-27

Analysis of Results: From the above, we can draw many interesting conclusions about the success of our bot. First, and perhaps most important, it tends to win. If the final results for each set of input files are summed, our bot gets more utility by far than the opponents. It does particularly well on the input file of size 7, never losing a single match up on that file. This could be just because of the size, it might be a "sweet spot" of available options, enough range for the bot to select mutually beneficial options without conceding too much. Another important takeaway is that overall the bot managed to come to mutually beneficial agreements. While it didn't win every time, it rarely got negative utility out of a matchup. This is very important, as non-cooperative agents will tend to get big payoffs sometimes, but will also occasionally fail and pay the penalty. By avoiding that penalty, we hope our agent will have a better overall score than others.

When we look at the data, we can draw some rough conclusions about why it behaves a given way against the bots it is facing. For example, we can tell that just by starting with a high threshold of utility it has to satisfy, it easily beats the random negotiator, by merely holding out for an offer better than the required minimum at a given step. It still does well against the bot that has that behavior of decreasing utility, we imagine this is because our bot notices the trend, and cooperates with it, but will also occasionally hold its ground or play tougher, which leads to better overall outcomes. If the test bot continues to decrease in trend, but the threshold is lower, our bot will naturally prevail even while cooperating. Our bot tended to lose or tie most often against the other bot that learns. They have very similar strategies though, so it isn't hard to imagine why they would have more similar behavior. Interestingly, the bot does fairly well against itself, often finding a mutually beneficial solution. In some

cases it did experience issues with not cooperating, but the random factor seems to add enough variance that it can find a “pretty good” compromise on most inputs.

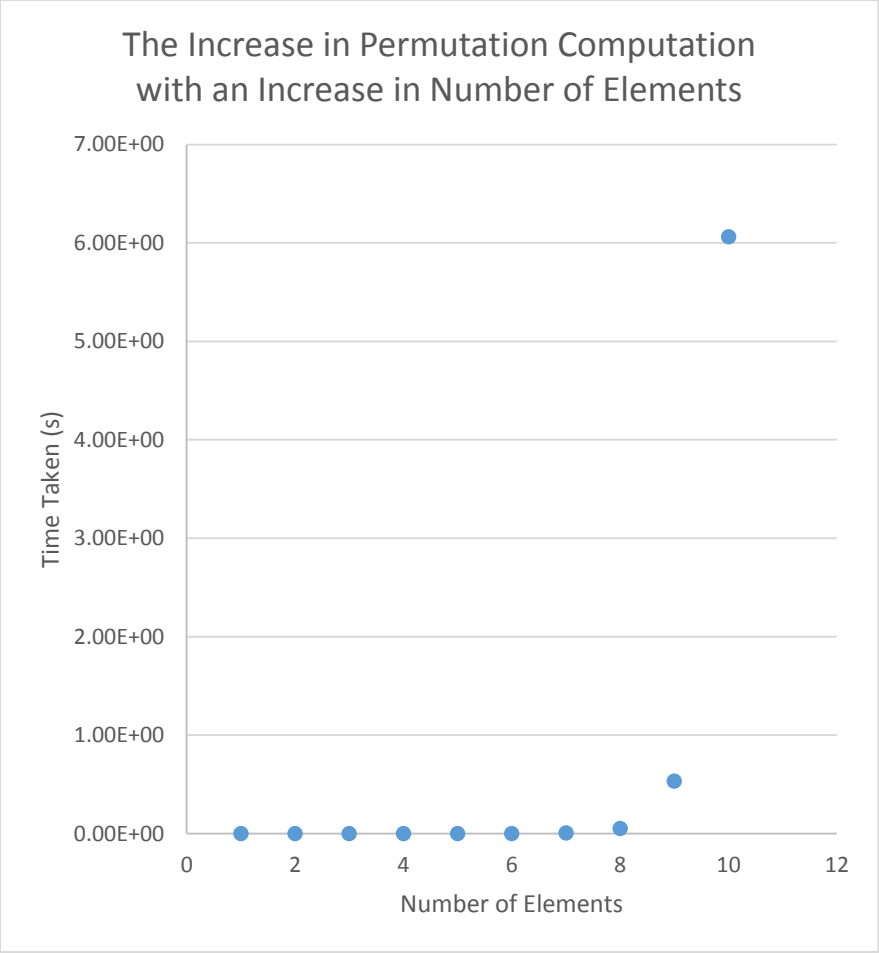
It is not terribly surprising, but it is important to note that going first or second has no noticeable difference on the scores. This is good in that the agent doesn’t rely on going first or second, which is not something that is within our control. This removes one limitation on our code, and was intentional, we didn’t want to depend on going first or second. Another intentional design decision was starting at the maximum utility and coming down. We could have started lower, certainly, but we were concerned by two primary possibilities. If we happened to select an offer that had somewhat low utility for us but great utility for the opponent, they might immediately accept and we would get a suboptimal deal. Similarly, based on how our agent adjusts its offers, if we faced a very stubborn opponent, we would have been outmatched in situations where there were only a few rounds per negotiation.

While in our tests for winning this was a non-issue, we found that this agent doesn’t tend to do well in situations where there is a large list and the two preferences are far apart. In this case, both agents will be likely to take a loss, as the best solution for each may still have negative utility. We were uncertain whether the solutions our agent found were actually close to the best middle ground, minimizing loss for both, or not. Even so, it was made clear by the instructors that this situation will not be the norm, so it is not unreasonable for us to consider behavior in these circumstances an unusual exception.

We probably should have had test cases that took a long number of rounds to see how the performance changed, instead of testing everything on 10. We did have a similar effect in testing one run of sample inputs at a time, or the 3 that we reported, which gave our agent a better chance to “learn” the opponent. We do not have the number to prove it, but qualitatively it seemed our agent was better in the long run, whereas when there is less time the behavior is more erratic.

Future Improvements: Based in part on the results of our tests, we had several ideas for how to expand on the agent that might help it beat certain opponents. We considered categorizing the data by the situations of the agent, for example, storing the probability of the opponent making an offer than increases their utility given that we increased ours on the last offer. Storing the information in such a way would probably look a bit like a Bayesian network, growing more complex depending on how many factors we consider as we collect information. We didn’t have time to implement this, but we believe it would allow us to make more accurate predictions about what our opponent will do, which in theory would allow us to better react.

Important Note: Both of these agents made use of Python’s itertool’s library’s permutation method. Since finding permutations is an exponential function, we decided to see how big a difference the number of elements for which we attempted to find a permutation made. The result can be seen in this graph:



From the chart, it is clear that there is a jump at the 10 - element mark. We attempted to find the time it would take for more elements, but unfortunately received a memory error. Given this finding, neither of the agents would work if given elements of size greater than 10 with the given time constraint of 30 seconds, assuming there was enough memory to store the various permutations (which we recognize may not be the case).