Name: Jordan Le
Date: 10/27/19
Email: jor25@pdx.edu

Hw1 Dinner Party Write Up:

Approaches:

- In my experimentation, I initially worked with randomly populating the table and continually randomizing to see if the score improved. This method was used as a performance baseline for an official AI algorithm. In the experimentation, it seemed like the random AI reached within the 80 ~ 100 range on the first instance, 200 ~ 285 on the second instance, and 10 ~ 40 on the third instance. All of these experiments were conducted in the span of a minute.
- The heuristic I used was the table's total score. If the table score improved, then keep a copy of that table. Otherwise, if it was lower, go to the next random table. It was a simple, but effective heuristic.
- I then went with a greedy initialization method by randomly placing an individual in the first seat and then selecting any of the top 40% optimal individuals that would sit opposite or adjacent to them. (This being the pool of individuals who gave the highest points when put together.) From there, I used the two placed individuals to identify the next optimal person in the bottom corner. After that, I went back to the top to select the next optimal individual to the right. Then repeated the step of using the top and bottom to select the bottom corner. This is shown in the figure below.

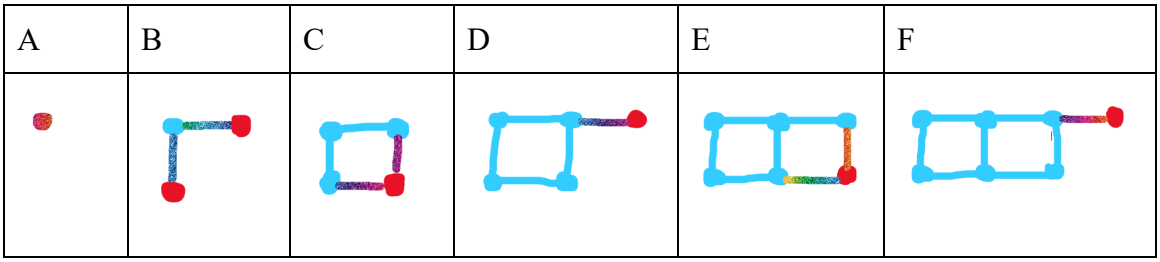


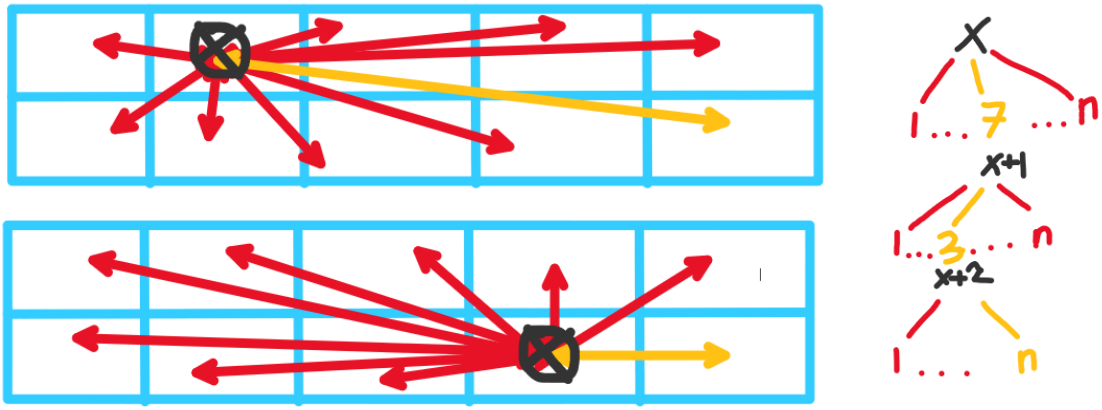
Figure 1: Greedy initialization. The vibrant color is a random element from the pool of optimal choices, red is the next element to be placed, and blue is the selected value. Read the figure from left to right and it shows how the first element is placed in the corner seat at random (A). Then from there, two optimal or suboptimal solutions are selected and placed (B). The next image shows how the two previously placed individuals are used to determine an optimal person to place in the bottom corner (C). The figure then continues on and moves one to the right based on the top corner (D). Then it repeats the cycle until the table is fully populated (E) (F). Hense prioritizing greedy placement based on the first element placed (A).

Local Search Agent:

- The local search agent takes in a (greedy) seated table and does a series of swaps for each of the individuals. It starts by calculating the initial score of the seated table. Then it goes

through each individual person, finding their indexes and switching them with each of the other people in the seating arrangement. (ie: px will switch with p(n-px))

- On each switch, it calculates the score to see if there was any improvement from the initial score calculation. If it does find that improvement, then it holds on to that configuration with the highest score heuristic. It then swaps the elements back to make sure the next switch is not altered by the previous.
- Next, it continues to search through all the remaining people and takes the most optimal solution. On completion, the highest sub-score and sub-table configurations are returned to compete with the previously initialized table scores. If this sub-table has a better score, the finalized table seating is updated. Otherwise, changes are discarded. Afterward, the cycle continues and a new greedy table is provided to the local search to optimize. Refer to Figure 2 below.
-



- Figure 2: Local Search Algorithm - The black X circle is the element being swapped. The red arrows show X switching with another element, but not the most optimal element. The yellow arrow shows the optimal switch for X. As shown in the images above, the specified element (X) is swapped with each other element by value, and then takes the best configuration of the swaps before going to the next element to repeat the process.
- Ex: Assume X = 1 through n, following the figures above, the element X is swapped with all values not equal to X to look for local score improvements. Once it goes through all values from 1 through n, it takes the table configuration with the greatest score. Then repeats the process for the next value of X, until X > n. (Shown in the branching image above) At that point, the table is compared to the best table score and either kept or discarded before a new greedy table is provided for another cycle.

Inspiration:

- Inspiration for this algorithm is based off of the hill climbing approach described in the Modern Approach 3rd Edition. It climbs quickly following the greedy heuristic, but has the potential for accidentally reaching local maxima instead of a global maximum. A possible idea to deal with local maximas in my algorithm is (optimally) randomizing after every individual has been moved. This is to potentially place the state in a position closer

to the global maximum. Further explanation for this approach is shown in the figure below from Russel and Norvig's Modern Approach 3rd Edition, page 121.

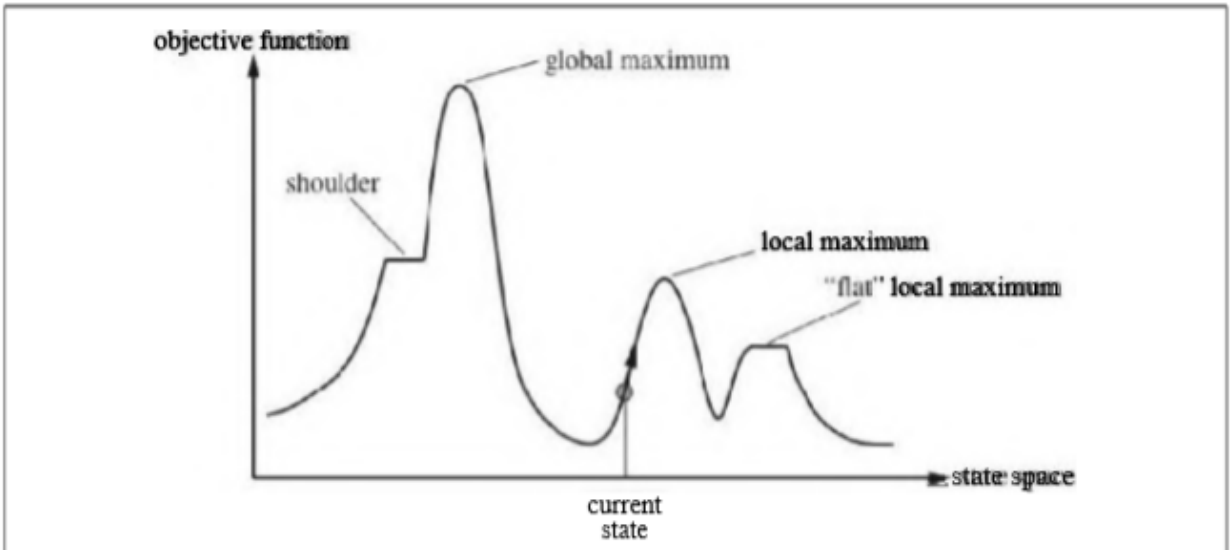


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

How to Run:

- I created a make file for running the program on each instance. Use the commands
  - "make run_inst1" - runs the greedy initialization with local search on instance 1.
  - "make run_inst2" - runs the greedy initialization with local search on instance 2.
  - "make run_inst3" - runs the greedy initialization with local search on instance 3.

Results:

- The results of the local search algorithm have consistently provided an optimal 100 on instance 1. I can also say that these results usually happen within the first 10 seconds or less. This is a significant improvement from the completely random solution which on occasion reaches the optimal 100 closer to the end of the 60 second trial period.
- On instance 2, the results range from approximately 470 ~ 510. If we compare those scores to that of the random baseline agent, 200 ~ 285, we see at least a 200 point improvement in scoring and a much higher consistency in score ranges.
- Lastly, for instance 3, the results range from 110 ~ 130. If we compare this to the random agent, 10 ~ 40, we can see the scoring practically tripled the upper bound. Overall, I found the local search algorithm to work quite well with the maximizing score heuristic. It definitely triumphs over the random AI, but more optimizations can be made to prune the search space and allow the AI to search more efficiently for the optimal answer.

Optimizations:

- An optimization I saw in the scoring function is the h(p1, p2) + h(p2, p1) is actually just the transposed preference matrix, element-wise summed, with the original preference matrix. Hence, doing this work before entering the loops significantly cut down on the number of steps for scoring each table.

Hardware and Software Set up:

- For software development and experimentation set up, I used a 64-bit Ubuntu virtual machine with a 2048 MB base memory. I worked on a virtual machine so I could experiment freely without needing to directly alter my device. I suppose I could have also used a virtual environment set up, but my computer had some confusion on which python versioning.

Resources:

- Used the following for reading in the matrix from the .txt files and loading them into 2d numpy arrays. https://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html
- Used for finding the top n elements in a numpy array. https://docs.scipy.org/doc/numpy/reference/generated/numpy.isin.html
- Element wise addition of the elements. https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.add.html
- Used to find specific values in the 2d numpy array. https://stackoverflow.com/questions/25823608/find-matching-rows-in-2-dimensional-numpy-array
- Used for fast switching indices. https://stackoverflow.com/questions/22847410/swap-two-values-in-a-numpy-array