Justin Lin
Alex Luo
Ashwinee Panda

170 Project Writeup

We worked with two algorithms designed to solve small and large problems respectively.

For the student generated inputs, we wrote our algorithm centered around the Numberjack Python module, which implements the miniSAT solver library. Our code parses the input into variables and constraints consistent with the Numberjack API function parameters. We express the constraint that Wizard3's is not between that of Wizard1's or Wizard2's through the expression: (Wizard3 > Wizard1 && Wizard3 > Wizard2) || (Wizard3 < Wizard1 && Wizard3 < Wizard2). Numberjack internally reduces this relationship into SAT clauses by generating additional variables. Although SAT solving is fast (which makes it ideal for problems with few variables and constraints), for larger files such as the staff inputs, the number of generated variables and clauses increases to the point where intelligent exhaustive search takes too long. The miniSAT algorithm running on 400 wizards and 4000 constraints generated 395600 variables. We realized that if we wanted to finish the project before the deadline, we would need a new strategy.

We eventually decided on local search, since we knew it would be infeasible to go through every possible permutation. We wanted to incrementally travel towards the optimal ordering, and we were aware of two potential implementations of this: simulated annealing and epsilon-greedy, local search. We became aware of simulated annealing when reading the documentation for the python simpleAI module, which had an example of utilizing simulated annealing quite simply. We tried and tried to make our algorithm work, but it simply wasn't converging. We dug deeper into the simulated annealing algorithm, attempting to modify some hyperparameters so that we could implement it ourselves, and found something puzzling. Simulated annealing had a chance to halt and do nothing -why? Well, we realized that simulated annealing in particular was written to be able to find a global minima when there was no strong certainty of the value of the function at the global minima; i.e. simulated annealing needed to have a chance to 'do nothing' in order to ever stop and not just oscillate around forever. However, this was not something that we needed to consider for this problem, because we knew that there was a guaranteed optimal ordering, and we could check whether our ordering was the optimal ordering in linear time. Therefore, we looked for some alternate algorithms that would allow us to 'perturb ourselves' out of the local minima we found ourselves in.

In particular we switched to epsilon-greedy local search (EGS), since it is possible to incrementally progress toward a good ordering. We start with a random configuration and make swaps to move us closer to our goal of satisfying all constraints. To ensure that we could lookup the positions of wizards quickly, we make a mapping from wizard name to index number. Then we store the problem state as a dictionary, which maps wizard index numbers to their position

in the current ordering. To store the constraints we use a list of 3-integer tuples, which describe the constraints in terms of the wizard index numbers (in which the 3rd wizard should not be in between the other 2).

Our initial version of EGS considered too many possible actions and took a long time to even complete a single update, so we decided to reduce the number of considered actions during each iteration. During each iteration of EGS, we generate our actions based on one of the currently unsatisfied constraints to ensure that we move towards our goal in at least some direction (even if we may get worse in other aspects). Our action space thus consists of 2 possible swaps we can make to satisfy the chosen constraint (swapping the middle item with one of the 2 boundary items). With epsilon probability, we take a random action from the two. With 1 - epsilon probability, we select the action that maximally reduces the number of unsatisfied constraints. With each iteration, epsilon is gradually reduced until it reaches a threshold: as we search toward what we feel is a good solution, we want fewer noise actions. However we want to make sure that epsilon is still large enough to allow us to escape a local optimum through random actions, so we stop decrementing it at a predetermined limit.

It is important to have an element of nondeterminism, since greedily satisfying as many constraints as possible during each interval will not necessarily lead us to the solution as it can lead to dead ends which may sometimes require us to take a non-optimal action in order to escape. Thus we ensure that our algorithm can make locally suboptimal actions by having an epsilon probability of taking a random action. Note that when we take this random action, although we satisfy one constraint, we allow other constraints to be broken if necessary. Also, we always ensure that our algorithm satisfies the chosen constraint even if it could lead to less constraints satisfied overall. This keeps our algorithm moving towards the goal and lessens the chance of it getting stuck in a local optimum and choosing not to advance.

It took some testing to find reasonably effective hyper-parameters (starting value of epsilon, how to decrease epsilon, and minimum epsilon threshold). We ran the EGS algorithm on smaller size problems and selected values that consistently gave us a solution in a reasonable amount of time. However, in the case that we choose an initial starting configuration that is so bad our algorithm cannot find an optimal ordering, we have also included a failsafe which restarts the algorithm on a new starting ordering after a certain (fairly generous) period of time.

It appears that even though our SAT solution is too slow for large input files, intelligent exhaustive search is better for the smaller files. Our theory is that fewer wizards translates to fewer correct orderings. Local search can get stuck down unviable action paths while SAT simply efficiently brute forces the problem. However we can usually arrive at a solution with our EGS algorithm for all the outputs in seconds, and a few minutes in the worst cases. All in all, we're quite proud of our algorithm as it can solve even the largest outputs consistently in a few minutes.

Installing libraries:

The only libraries we used in our code were Numpy and Numberjack.
Numpy:

- Numpy is good for random numbers, which is what we mainly used it for
- To install Numpy, run: pip install numpy.

Numberjack:

- We decided to use it as it allowed for an easy (albeit slow) implementation of an SAT solver. It would allow us to input constraints and output the proper ordering.
- To install Numberjack, run: pip install Numberjack.
    - Numberjack is Mac only (so just run EGS on Windows)
    - Our Numberjack solution should be run using Python2.7 (install Numberjack for python2.7

Running our algorithm:

- To run our Numberjack solution:
    - For student inputs: python 170_solver.py
        - The output will be saved in .txt files (runs on all input files)
    - For staff inputs: python 170_solver.py <number of wizards in file>
        - The output will be saved in a .txt file (only runs on one input file)
- To run our Epsilon-Greedy solution: python epsilon_decreasing.py <name of input file>
    - At the end the program will print out an optimal ordering of wizards, which can then be copied to the desired output file.