Wizardry Party Phase II Writeup

Xintong (Robert) Wen & James Scott Reese

Our base algorithm reduces an instance of the wizardry party to become an instance of an ILP, another NP-complete problem. We have a python script `wizard_ordering.py` writes and executes an LP script.

Each wizard becomes a Integer LpVariable with possible values ranging from `1` to `n = num_wizards` because `num_wizards` would be the minimum number of integer values required to properly satisfy all constraints. We use `bigM` and `smallC` constasnts to force strict inequalities. Each constraint also makes use of a helper binary variable `z` that indicates on which end of a range a wizard lies. Using the constraint `Tony Bruce Thor` as an example, we would have the following lines in our script:

The following block would have been executed before iterating through the constraints:

```
smallC = 1
bigM = 5 + num_wizards
Tony = LpVariable("Tony", 1, num_wizards, cat="Integer")
Bruce = LpVariable("Bruce", 1, num_wizards, cat="Integer")
Thor = LpVariable("Thor", 1, num_wizards, cat="Integer")
```

Once we have reached this constraint through our iterations, the following block is executed:

```
z = LpVariable("z", cat="Binary")
prob += Thor + smallC <= Tony + bigM * z
prob += Thor + smallC <= Bruce + bigM * z
prob += Thor >= smallC + Tony - (1 - z) * bigM
prob += Thor >= smallC + Bruce - (1 - z) * bigM
```

This imposes the constraints that if `z = 0` then `Thor < Tony and Thor < Bruce` and if `z = 1` then `Thor > Tony and Thor > Bruce`

We Use the PuLP package which has documentation at `https://pypi.python.org/pypi/PuLP/1.6.8` and can be installed as simply as running `pip install pulp` from the command line. We also tried a number of LpSolvers that PuLP calls, but the one with the best performance was the Gurobi solver. Gurobi offers a free academic license here `https://user.gurobi.com/download/licenses/free-academic` and comes with an easy to use installer. We were able to use the Gurobi Optimizer to solve up to inputs of size 140. For input sizes above 140, the Optimizer would take far too long and far too many resources so we instead used a randomized approach called simulated annealing for greater input sizes.

The basic idea for simulated annealing can be found here: `katrinaeg.com/simulated-annealing.html`. Our implementation can be found in `simulated_annealing.py`, which is called by `wizard_ordering.py` as a module when dealing with

larger input sizes. This method is non-deterministic because it uses randomization, but it is much faster. We essentially used this approach to brute force larger input files. The idea is to output a random ordering and then continuously improve upon it by comparing an ordering to its neighbor, which is a swapping of one wizard's place to that of another. If this reduces the number of failed constraints (as the `cost_opt` function) then we proceed with the new ordering, otherwise we may or may not proceed with this ordering based on an `acceptance_probability` method that compares the previous and current number of constaints failed. The number of iterations performed is dependent on the paramters `alpha` and `i`. The greater these values were, the longer the algorithm would run for, but the more likely we would return an ordering that satisfies all constraints. We noticed that when we used a list, we never cared about anything except the index of a wizard, so we improved runtime by operating only on a map of a wizard to its rank. For example, `Tony : 0` in the map would mean that `Tony` is the left-most wizard in the ordering.