Tindar: the optimal matchmaker

June 8, 2020

1 Motivation

I worked on this personal project for the Capstone project of Udacity's Machine Learning Engineering course. Although I liked working with SageMaker during the course, I felt like too much functionality was abstracted away. To really sink in the deployment process, I created Tindar, a completely custom Operations Research project that I came up with myself.

Tindar is now live at https://tindar-engine-xs-chx6ixua2q-ew.a.run.app! The web application is self-explanatory. This report describes the whole process of the project, from ideation to deployment. The full project can be found on my Github.

2 Domain Background

Suppose you owned a dating company. Once every week, people come to your venue to find their ideal partner. During the speed-dating round, every person expresses if he/she would like to meet his/her date in real-life in the next week. Your people have busy schedules, so they can only go on a real date with a single person.

Current dating apps are only focused on the individual, but Tindar tries to achieve the maximum happiness among the user community as a whole. Our job is to make your community as happy as possible by pairing up as many users as we can.

We can only pair people if there is mutual interest. Tindar is conservative: a person can only be paired up with one other person at most.

3 Problem Statement

The goal of this project is to build an optimization model that pairs up as many users as possible. A match can only be made if both users have expressed

mutual interest. As mentioned above, a user can only be coupled to one other user at most.

In this project, we build a mathematical and consequently a data representation for the problem described above. We then investigate how the performance of our model is affected by the size of the Tindar community and the number of people inerested in each other. We focus on the following questions:

1. Can we always find an optimal solution, even for large Tindar problems? If not, can we find a heuristic that provides a decent solution? How far is the heuristic's solution from the optimal solution?

2. How is the computation time affected by the size of the Tindar problem?

4 Datasets & Inputs

We represent the tindar community as a binary square matrix, where a 1 at position (i,j) represents that person i is interested in person j. Because this problem is fictional, we do not know this interest matrix. Therefore, we will randomly generate it. The data generation is thus part of the project.

The main concept is to generate this matrix based on a Bernouilli distribution (flipping a coin). This distribution is characterised by parameter p, the probability of one user expressing interest in another user. To make an easier interface to the user, p is controlled by the parameter 'connectedness' according to the scale:

$$p = \frac{connectedness - MINCONNECTEDNESS}{MAXCONNECTEDNESS} \tag{1}$$

where MINCONNECTEDNESS=1 and MAXCONNECTEDNESS=10. The user thus inputs a number between 1 and 10 instead of a probability.

To simulate a Tindar problem, do:

1. set n, the number of members in the community

2. set the parameter 'connectedness'

3. calculate the Bernouilli parameter p

4. sample $n \times n$ elements from the Bernouilli(p) distribution

5. reshape the samples into an $n \times n$ matrix

In this way, we can generate as many Tindar datasets as we like, which supports our research goals.

5 Solution statement

Mathematically, the Tindar community can be seen as a graph, where each node represents a user, and a directed edge the interest of one user in another. The model tries to pick as many edges as possible without violating the constraints (mutual interest, max. 1 partner).

This maximum pairing problem can be formulated as a Binary Linear Program. Let $(a_{i,j}) \in \mathbb{R}^{n \times n}$ be the interest matrix consisting of ones (if one user is interested in the other) and zeros (else). Then the model needs to decide which users to pair-up with the decision variables $x_{i,j}$:

$$x_{i,j} = \begin{cases} 1, & \text{if user i is coupled to user j} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

5.1 Objective

The objective is simple: to pair up as many users as possible.

$$\max_{x_{i,j}} \sum_i \sum_j x_{i,j} \tag{3}$$

5.2 Constraints

There are a couple of constraints that the solution needs to obey.

The first is obvious: users can only be paired if they are mutually interested in each other.

$$x_{i,j} \leq a_{i,j} \quad \forall i, j \tag{4}$$

Next, pairing is obviously symmetric: if user i is paired to user j, then the reverse holds as well.

$$x_{i,j} = x_{j,i} \quad \forall i = 1, 2, ..., (n-1); j = (i+1), i, ..., n \tag{5}$$

We also have that a single user can only be assigned to 1 other person (one outgoing arc at most), and a user can be assigned to only a single other user (one incoming arc at most).

$$\sum_{j=1}^{n} x_{i,j} \leq 1 \quad \forall i = 1, 2, ..., n \tag{6}$$

$$\sum_{j=1}^{n} x_{i,j} \leq 1 \quad \forall i = 1, 2, ..., n \tag{7}$$

5.3 Full model

We thus have the Tindar model:

$$\max_{x_{i,j}} \quad \sum_i \sum_j x_{i,j} \tag{8a}$$

$$\text{subject to} \quad x_{i,j} \leq a_{i,j} \qquad \forall i, j, \tag{8b}$$

$$x_{i,j} = x_{j,i} \quad \forall i = 1, 2, ..., (n-1); j = (i+1), i, ..., n \tag{8c}$$

$$\sum_{j=1}^{n} x_{i,j} \leq 1 \qquad \forall i = 1, 2, ..., n, \tag{8d}$$

$$\sum_{i=1}^{n} x_{i,j} \leq 1 \qquad \forall j = 1, 2, ..., n \tag{8e}$$

$$x_{i,j} \in \{0, 1\} \qquad \forall i = 1, 2, ..., n \tag{8f}$$

Note that all constraints are linear and the decision variables are binary. We are thus in a Binary Linear Programming setting. Such problems can be tackled with algorithms like Branch & Bound, however for this project we will use libraries that are ready to go. I have never properly experimented with the size of BiLP problems that todays software (CPLEX, Gurobi) is able to tackle, but my estimate is that the solvers should have no problem whatsoever with $n = 100$.

5.4 Python Implementation

I used OOP to encapsulate the Tindar problem, its corresponding BiLP representation, and the methods for solving the problem. The model can be built and solved in Python using the PuLP library. We will use the default PuLP solver.

I also wrote a small class to generate Tindar problems as described in Section 4. The Bernouilli draws can be done with Numpy.

6 Benchmark model

To benchmark the PuLP solution I wrote a greedy heuristic, which solves in $\mathcal{O}(n^2)$. This heuristic works as follows:

for i=1, ..., n

- get the i'th row of interest matrix A, which is the people that person i is interested in. If person i is already assigned to someone else, go to i+1. Else, initialize match_made=False, j as small as possible with $j \neq i$. Then, while not match_made:
  1. if $a_{i,j} = 1$ and $a_{j,i} = 1$, set $x_{i,j} = x_{j,1} = 1$. set match_made=True
  2. else, $j = j + 1$ (skip $j == i$)

This solver was added as a method to the Tindar object.

All of the source code relating to the concepts above can be found in ./tindar-engine/tindar.py (see my Github).

7 Evaluation metrics

The main evaluation metric is of course the optimisation objective (the number of people paired). We also look at computation time.

8 Results of Tindar experiment

In the experiment, we generate a series of Tindar problems (characterised by the A matrix) to investigate the solution behaviour (characterised by objective value and solution time) w.r.t. n, the number of people in the Tindar problem, and the number of edges in the graph (referred to as connectedness).

8.1 General Experiment

All computations are performed on my local machine (HP Zbook with 8 intel i-7 cores and 32GB of RAM). For the general experiment, I used the following parameters:

- n_list = [10, 30, 50, 100, 200, 300, 500]
- connectedness_list = [1, 3, 5, 8]
- repeat = 10

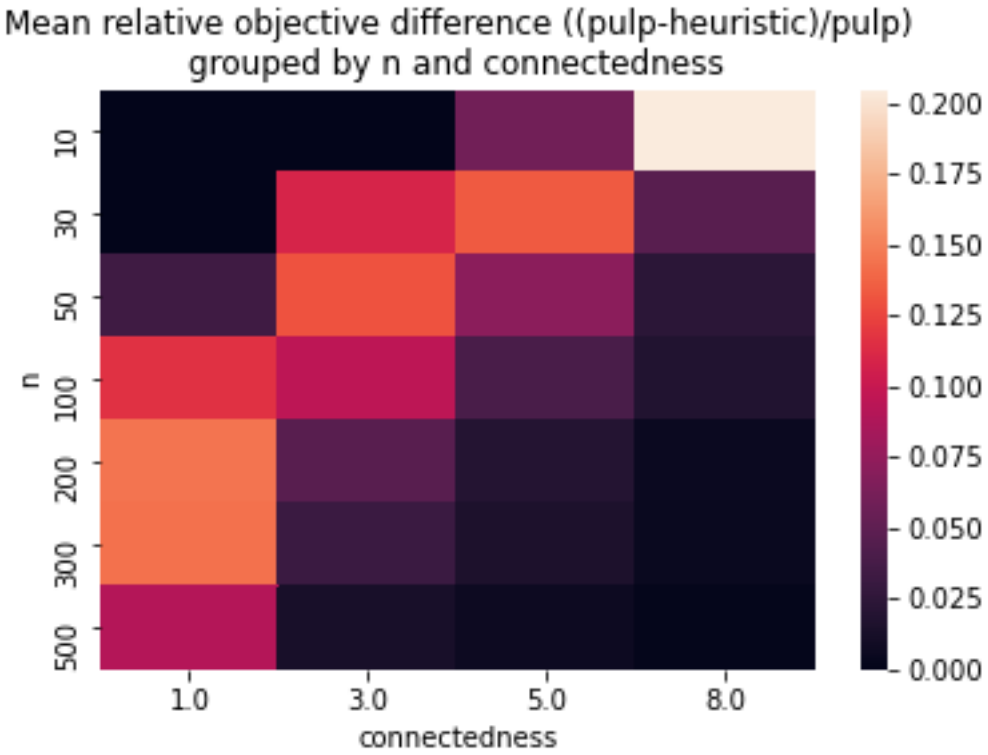
I then created the Cartesian product of n_list with connectedness_list to initialize Tindar instances, yielding (n, connectedness) pairs: [(10, 1), (10, 3), ..., (10, 8), (30, 1), (30, 3), ..., (500, 8)]. For every item in this list, I generated repeat=10 random Tindar problems for statistical comparison. I then solve that instance with both PuLP and the heuristic, and write the results to a json file in ./data. You can do it yourself by executing tindar.py as main script. The statistics and graphs are obtained from the notebooks.

8.1.1 Difference objective value between PuLP and Heuristic

For these parameters, PuLP was always able to solve the problem to optimality. We look at the relative difference in objective values (recall, the objective is the number of people paired):

$$\text{rel\_diff} = \frac{obj_{pulp} - obj_{heuristic}}{obj_{pulp}} \tag{9}$$

In the heatmap below we can inspect the relation between this relative difference, n, and connectedness:



For small connectedness, the greedy heuristic is on average more than 10% off. For large connectedness, the greedy heuristic comes up with a near-optimal solution. In general, we observe a negative correlation between connectedness and the relative difference, which makes sense. If the graph is nearly fully connect, it is easy to find a solution, since you can pair any user with almost any other user.

8.1.2 Difference solution time between PuLP and Greedy Heuristic

The solution time for PuLP includes the creation of the PuLP constraints (for the heuristic this step is not necessary)

Mean solution time per n for PuLP:

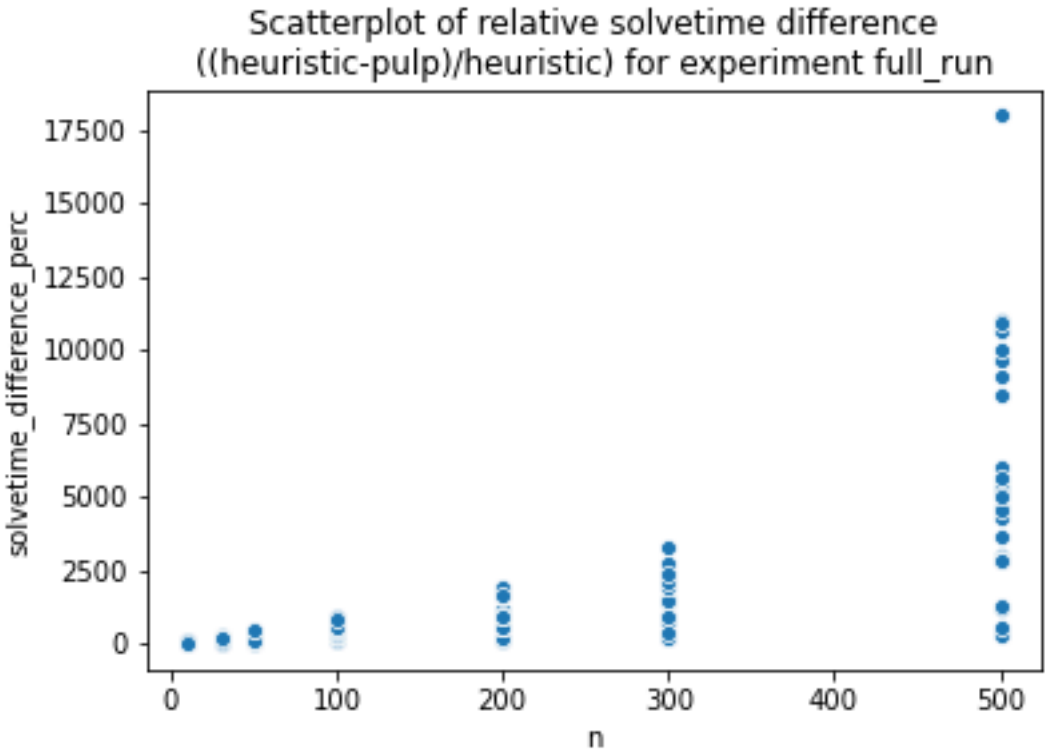
n	solution_time (s)
10	0.042812
30	0.129114
50	0.307569
100	1.201604
200	5.055357
300	12.424393
500	79.592850

Mean solution time per n for heuristic:

n	solution_time (s)
10	0.000840
30	0.001256
50	0.002086
100	0.004469
200	0.010853
300	0.019004
500	0.035000

The heuristic is significantly faster.

Similar to the objective value, we can calculate the relative difference in solution time between PuLP and the heuristic. In this case, we reverse PuLP and the heuristic because the heuristic is faster. The relative solution time difference displays a seemingly exponential relationship with n.



For $n = 500$, the heuristic can be >10,000 times faster than PuLP.

9 Conclusions of Tindar Experiment

- Both PuLP and the heuristic can solve reasonably large Tindar problems (up to $n = 500$) well.
- PuLP's optimality guarantee comes with the cost of increased computation time. PuLP does not seem massively scalable, but for the use-case described in the introduction this is not necessary.
- The heuristic provides a sub-optimal solution for cases where the connectedness is low. The heuristic's objective value for cases where the connectedness is high is nearly optimal. The heuristic is fast and could be applied to cases where n is large.

10 Possible refinement

The solution implemented is already pretty mature (including type checking for instance creation and a simple test script with pytest), but could be improved by implementing parallel computation for the main tindar.py script.

11 Deployment

11.1 Web Application

To launch my project, I built a minimalistic but powerful web-application with Flask, that allows a visitor to :

- view information about the Tindar project on the home page
- learn how to interact with the API
- generate a Tindar problem and have its representation sent back to him
- post a Tindar problem and have the solution together with other relevant information sent to back to him
- download this report

Both API endpoints send back JSON data. For more information on the API, check out https://tindar-engine-xs-chx6ixua2q-ew.a.run.app/api. To make the app intuitive, I wrote two simple HTML's (one for the homepage, one for the api) that explain about the Tindar project (for the first and second item).

11.2 Platform

In my proposal, I stated that I wanted to deploy on AWS using AWS Lambda and API Gateway. During the development however, I jumped straight into Flask without checking how that might be pushed to Lambda. When the app was nearly finished, I came up with two solutions:

- Break every endpoint of the Flask app, and deploy every endpoint to a single Lambda function.
- Use the zappa library to push

However, because I did not want to waste time learning another technology or completely refactoring my code I decided to Dockerize the app and serve it with Gunicorn. I still had a GCP cloudbuild script lying around that automatically pushes an image to GCP Container registry, so I decided to use Cloud Run (serverless container deployment) to serve the app. Cloud Run comes with a designated url automatically. To safeguard against high cost, I put a restriction on the size of the Tindar problems that can be solved ($n <= 50$).