Introduction

In this project, we consider the housing market problem from a variety of perspectives, both theoretical and practical. In the simplest formulation of this classic problem, a set N of agents owns a set H of houses, and the goal is to find a matching that is Pareto-optimal. Roughly speaking, this implies that under the matching scheme, there is no deviation benefiting an agent i that would not decrease the utility of an agent $j \neq i$, for all i and j.

The problem becomes more interesting when a certain amount of leeway is granted with respect to the parameters of possible solutions. By opening the door to probabilistic algorithms, tweaks to the initial ownership schema, and permutations of preference generation, some very interesting results can be derived, especially when the metrics used to discuss solutions are broadened beyond simple Pareto-optimality.

For our project, we were interested in programming three of the most important algorithms associated with the house allocation problem, in addition to developing software to analyze the efficiency and quality of the solutions output. In particular, we created programs in the python language to model the Top Trading Cycles, You Request My House I Get Your Turn, and Probabilistic Serial algorithms, hereafter referred to simply by their (perhaps truncated) initials (TTC, YRM, PS, respectively). In addition, we analyzed the solutions returned by these procedures via metrics including ex-post Pareto-optimality, time complexity (performance speed), and envy-freeness. Last, we played with preference generation, considering the Mallow model but settling on a scheme relying upon cardinal utilities modeled from a hierarchical normal distribution, guaranteeing correlated preferences. While our writeup will highlight our findings and showcase the most important pieces of our project, we stress that the brunt of the effort expended on this collaboration went into an extensive implementation of the several algorithms and solution metrics, the entirety of which is included with this paper. At certain points in the paper, we will direct the reader to run the scripts associated with certain files if so desired, to see results firsthand. In addition, we will often provide sample output within the text so as to avoid extensive context-switching.

In the first section, we will discuss the preference generation techniques utilized to parametrize our agents' utilities. Next, we will give an overview of the algorithms that we coded. Following that, we detail the metrics used to analyze the quality of the solutions we generate. We then move to our experimental results, showing sample

inputs and outputs to our algorithms. Ultimately, we discuss the challenges we faced and address some of the reasoning behind our design decisions in our conclusion.

Preferences

At the heart of all matching problems is a preference scheme that dictates each agent's preferences. In particular, a preference system must be able to determine, given two alternatives, which, if either, is preferred. In this instance, we model pairwise preference as a cardinal comparison between the utilities afforded to an agent by the two alternatives proposed. More formally, an agent i's utility over alternative a in set A is given by a function $u_i : A \to \mathbb{R}$, and an operator such as $\succ$ is defined (loosely) by the equivalence $u_i(a_1) > u_i(a_2) \equiv a_1 \succ_i a_2$. In relation to our problem, the preference operator denotes that an agent prefers the assignment of a particular house (or other item, though housing is the canonical example in the literature) to that of another.

Thus, the way in which we model preferences inherently dictates the scope, direction, and results of our simulation. One option we explored early on (at the suggestion of Debmalya Mandal) was the Mallow model, a way of generating ordinal (and thus pairwise) preferences across a set of alternatives, clasically parametrized by a dispersion factor ϕ that relates the "closeness" of agents' preferences. In other words, dispersion is the degree to which there is an underlying "true" quality assessment that each agent tends to agree with. However, we decided against this model because preferences in the real world are poorly modeled by way of randomly selecting alternatives to be better than others, even if there is a system to the randomness. Under the Mallow model, there is no way to control the degree to which a house is more inherently desireable than another. However, a system of two-tiered normally distributed preferences allows a great deal of control over both the underlying quality distribution of houses as well as the degree to which agents' preferences take into account this distribution.

We model preferences as follows. Build a vector $V \in \mathbb{R}^{|N|}$ by sampling from a normal distribution with mean μ and variance σ^2. For each v_i in V, create a normal distribution with mean v_i and variance η^2. For each agent, sample their utility for assignment of house i from the secondary distribution. In this way, one can fine-tune the way preferences correlate between agents. On one side, increasing σ spreads out the underlying qualities of the houses, making preferences more distinct, and thus less independent. On the other, increasing η makes differences in underlying quality

less significant in determining pairwise preferences. Providing a snippet:

```
def assign_preferences(n=config.NUM_AGENTS):

    # create means
    item_means = [Distribution(config.ITEM_MEAN, config.ITEM_VAR).sample()
                  for _ in xrange(n)]

    # create agents and shuffle their order
    agents = [Agent(i) for i in xrange(n)]
    random.shuffle(agents)

    # assign agents preferences

    for agent in agents:
        agent.cardinal_prefs = [Distribution(
            item_mean, config.PREFERENCE_VAR).sample() for item_mean in item_means]

    # sort agents' preferences
    for agent in agents:
        agent.ordinal_prefs = [sorted(agent.cardinal_prefs).index(x)
                               for x in agent.cardinal_prefs]
```

When run with standard configuration

```
ITEM_VAR = 63 # sigma^2
ITEM_MEAN = 50 # v_i
PREFERENCE_VAR = 44.7 # eta^2
NUM_AGENTS = 100
```

the correlation between two agents' preferences is around 0.7, which we determined to be a sound basic representation of truth (see line 30 of sim.py).

Under this scheme, preferences are extremely parametrizable, and we are able to deliver more robust results from our algorithms by testing them on various preference sets.

Algorithms

We first describe and implement Gale's celebrated *Top Trading Cycles* algorithms, henceforth called TTC. TTC is a canonical algorithm for *housing allocation with existing tenants* problem. More specifically, there are n agents and n items (or houses). Each agent has a strict preference over all the items. Every agent can own at most one item. Before the algorithm starts, every agent owns exactly one item. After the algorithm, every agent also owns exactly one item, which may or may not be the same item it started with.

In English, the TTC algorithm goes as follows. In each round, each agent points to the agent that owns the most preferred item among all the remaining items. If, in any round, a number of cycles form, we perform the trade on each cycle. A *cycle* is when agent i_1 points to agent i_2, agent i_2 points to agent i_3, ..., agent i_{k-1} points to agent i_k. agent i_k points to agent i_1. We perform a trade on this cycle by assigning each agent the item she points to in the cycle, that is, assign agent i_1 with the item owned by agent i_2, and so on. Then we remove all agents and items involved in the cycle trade. Repeat until there are no agents left.

The name Top Trading Cycle mechanims reflects the way TTC works. At each step, the items on the top-preference cycles are traded, that is, each agent that trades gets her most preferred items among the available ones, and only the agents in the top-preference cycles trade in each round. A cycle can be a self-loop if $k = 1$, that is, the agent trades with itself and remove itself from the mechanism, or a k-cycle for $k > 2$ in which case the cycle trade described above applies.

It is not immediately evident that the above verbal description of TTC provides a well-defined algorithm. Does it always terminate? Are the top-preference cycles well defined? To guarantee that, we need anxillary mathematical results. We can prove (for example, Lemma 12.1 in lecture 12) that the directed graph formed in each round of the TTC mechanism includes at least one cycle, and when multiple cycles form in a round they are vertex-disjoint. The existence of a cycle in each round guarantees that the algorithm terminates in at most n steps, since a cycle means at least one agent is removed in that round, and there are n agents. The vertex-disjoint property of the cycle means that in each round no agent is traded in more than one cycle, so the trade-around-the-cycle mechanism is well-defined.

We also use the above mathematical result to aid our implementation of TTC, which is in a file called `algos.py` We keep track of the remaining agents (not yet traded) in `agents_remaining` and we repeat the round until all agents are traded, that is, `agents_remaining` is empty. In each round, we first identify which agents are in a cycle and which are not. This is well-defined by the above result. The

categorization begins by putting agents into three categories (variables in the implementation) `isCycle, isNotCycle, unknown`. At first, all agents are in unknown. Then, we pick an agent, say A0, from unknown, then we see what agent A0 points to (the agent that owns A0's most preferred item among the available ones), say A1, and so on. We keep track of the chain of agents in `cycle_track`, adding one agent at a time. So, for example, `cycle_track` is [A0, A1, A2] means A0 points to A1 and A1 points to A2. Each agent can point to one, and only one, agent among the available ones so this is well-defined. Since there are a finite number of agents, eventually the last agent will point to an agent that already exists in `cycle_track`, say [A0, A1, A2] and A2 points to A1. We have found a cycle. The repeated agent until the end of `cycle_track` is a cycle while all the agents before that are not in a cycle. We track this with`start_index`. In the example above, A1 is a repeated agent and appears in index 1, so `cycle_track`, starting from index 1, that is, [A1, A2] is a cycle, and agents before index 1, that is, A0, is not a cycle. Then we remove all agents above from `unknown`. Repeat until `unknown` is empty, and all agents are correctly categorized in this round whether they are in a cycle or not. Then we perform trades on top-preference cycles and remove them from `agent_remaining` while those in `isNotcycle` remain for the next round.

The final algorithm we implement is the Probabilistic Serial mechanism, a special case of the group of algorithms known as "Eating Algorithms", a name that fits very nicely! The algorithm involves a unit block of time, where at each $t \in [0, 1]$, each agent is "consuming" some probability of acquiring some item, where the item they are each eating is their top-rated, available (i.e., not totally consumed) item.

In our implementation[1], we model continuous time as arbitrarily resolute time-slices, and we denote each slice as a *round*.

Each item has a corresponding number of "morsels" that it has left (line 15): this denotes implicitly the fraction of the item's probability that is "left". In each round, an agent takes one bite out of the item that they most prefer of the set of items reminding, i.e., they acquire one morsel of the item that they are most partial towards, from all the items that still have morsels. In each iteration, the order of agents is shuffled (line 52; so as to eliminate the possibility of a corner case wherein there are several instances when each of n agents wants to eat the same item, with fewer than n morsels remaining, benefiting the first few agents in the order). At the end, the probability distribution is output (line 82; 85) in either fractional or decimal form,

[1]NB: For the remainder of the section, we will call the reader's attention to various line numbers in the file prob_serial.py.

and we check that the resulting matrix (rows of houses, columns of agents) is doubly stochastic, i.e., all the rows and all the columns sum to 1. This is the expected behavior, because assuming well-defined preferences over every item, and that each agent owns a house to begin with, the probability that an agent gets no house should be 0, and the probability that a particular house is not allocated should be 0. (To see output for PS, run the procedure *test_ps()* from the file.)

We note that moving from this probability matrix to an allocation is a trivial matter of flipping a biased coin and allocating each house sequentially, and is not implemented in our project, as most outputs of PS that we've seen in the literature is the doubly-stochastic matrix rather than an outcome drawn from that distribution.

Metrics

Experimental Results

Conclusion