CSC148 Assignment 2: Intergalactic Venture Planning

Due on July 9, 2021 at 23:59 EST

Contents

Coding Guidelines

All code that you write should follow the Function Design Recipe and the Class Design Recipe. You may assume that all arguments passed to a method or function will satisfy its preconditions.

For class `DistanceMap`, all attributes must be private. For all other classes except `Passenger` and `SpaceBike`, you **must NOT**:

- change the interface (parameters, parameter type annotations, or return types) of any methods provided in the starter code. Exception: you may change the inherited initializer for subclasses of `FleetScheduler`.
- change the type annotations of any public or private attributes provided in the starter code.
- create any new public attributes.
- create any new public methods except to override the abtract `schedule` method that `GreedyScheduler` and `BogoScheduler` inherit.
- add any import statements to your code.

You will be designing classes `Passenger` and `SpaceBike`; thus:

- You are allowed to make some or all attributes of `Passenger` and `SpaceBike` public.
- You may add public methods to `Passenger` and `SpaceBike`.

In general, you may:

- remove unused imports from the Typing module, or add imports from the Typing Module.
- create new private helper methods for the classes you have been given.
  - for any private methods you create, you **must** provide type annotations for every parameter and return value. You must also write a full docstring for such methods, as described in the Class Design Recipe.
- create new private attributes for the classes you have been given.
  - if you create new private attributes you **must** add the type annotation and include a description of them in the class docstring as described in the Class Design Recipe.
  - Exception: In class `PriorityQueue` we have defined all the attributes needed. Do not define any new attributes, even private.

Background

The year is 2067. You (and your assignment partner, if you choose to have one) are the (co)founder(s) of ZeroG Enterprise. ZeroG Enterprise is a cutting-edge ride-sharing service that offers interplanetary travel on enterprise-owned, completely autopiloted SpaceBikes.

SpaceBike Type	Max # Passengers	Fuel Capacity (ML)	Fuel Usage Rate (ML/AU)	BCO (M$)
Atom Bike	2-4	3.5-4.5	0.4-0.6	1.2
Nova Bike	5-20	6.7-8.5	0.8-1.2	2.4
MC^2 Bike	25-30	25.2-36.5	4.5-5.1	10.1

Table 1: Different types of SpaceBikes and their characteristics.

ML: MegaLiter (1 million liters)
AU: Astronomical Unit
BCO: Base Cost of operation, in millions of dollars (M$).

There are different SpaceBike models, each with their own characteristics, indicated in Table 1. Every type of SpaceBike has its own range of fuel capacity, range of fuel usage rate, and range of maximum Passenger capacities. For example, any given Atom Bike can have a fuel capacity of any floating point number between 3.5 and 4.5 ML. The fuel usage rate is independant of number of Passengers onboard, and is fixed for the lifetime of a particular SpaceBike. Note that the Fuel Capacity, Fuel Usage Rate, and maximum Passenger capacity differ within SpaceBikes of the same type. The base cost of operation is always the same for a given type of SpaceBike.

A SpaceBike always starts off with full fuel (i.e. it has exactly the amount of fuel as its maximum capacity). Once the SpaceBike starts travelling, it can not refuel.

Each location in our Galaxy has exactly one location that SpaceBikes dock in and embark from. The distances between these hubs on each planet is provided in the Galaxy Data file that you will read from in one of your tasks.

The SpaceBike business is so lucrative that the Passengers bid on fares, vying for a chance to experience ZeroG. Your task is to devise a program to plan the SpaceBike travel itinerary to attempt to maximize the profits you make, while taking into account those pesky physical constraints like Passenger capacity and fuel usage regulated by the authorities.

ZeroG Enterprise previously relied on manual scheduling of Passengers. However, as you amassed more SpaceBikes in your SpaceFleet, and the number of Passenger bids on space travel skyrocketed, it is no longer reasonable to do the scheduling manually. You will be writing a program to automatically schedule Passengers on SpaceBikes. Since you can think of multiple ways to prioritize Passengers, you would like some concrete numbers to direct the decision of how to schedule the Passengers to maximize your profits.

Preparation

1. Read the entire A2 handout and the provided starter code.
2. In Appendix B, the first 4 iterations of the greedy algorithm are demonstrated. Complete the remaining 2 iterations.
3. Complete the not-for-marks quiz.

Starter Code

Download the starter code a2.zip. Any files that start with `a2_` are files that you will need to modify and submit. Once you are done all the tasks, you will be able to run the file `brute_force.py`. `brute_force.py` runs the two schedulers you will create and graphs their relative performance. You are provided with a basic set of starter tests, `starter_tests_a2.py`, to run locally, and a moderate set of tests on MarkUs. You are expected to write your own test cases to thoroughly test your code, as we have additional hidden tests you will be evaluated on.

Tasks

This is a summary of the minimum requirements of each task that you are required to complete. This checklist is meant to be a guide to help you keep track of your work, but is not a comprehensive or complete description of your tasks.

- Task 1: `a2_distance_map.py`
  - define class `DistanceMap`
  - complete method `distance`
  - complete method `add_distance` consistent with doctest examples in `a2_space_bikes.SpaceFleet`
- Task 2: `a2_space_bikes.py`
  - define class `SpaceBikes` (including interface design)
  - define class `Passenger` (including interface design)
  - implement class `SpaceFleet`
- Task 3: `a2_container.py`
  - implement class `PriorityQueue`
  - complete method `is_empty`
  - complete method `add`
  - complete method `remove`
- Task 4: `a2_fleet_scheduler.py`
  - define class `BogoScheduler`
  - complete method `BogoScheduler.schedule`
  - define class `GreedyScheduler`
  - complete method `GreedyScheduler.schedule`
- Task 5: `a2_command_central.py`
  - complete function `load_fleet_data`
  - complete function `load_galaxy_data`
  - complete function `load_passenger_data`
  - complete class `CommandCentral`
    - method `init`
    - method `_compute_stats`
    - method `_print_report`

Task 1: Distance Map

Task 1 is completed in `a2_distance_map.py`.

Use the Class Design Recipe to define a class called `DistanceMap` that lets client code store and look up the distance between any two galaxy locations. Your program will use an instance of this class to store the information from a map data file.

If a distance from location A to location B has been stored in your distance map, then it must be capable of reporting the distance from location A to location B and of reporting the distance from location B to location A (which could be the same or different). You may wish to refer to the Galaxy Data file, as this is the data that your `DistanceMap` will represent.

Your `DistanceMap` class *must* provide a method called `distance` that takes two strings (the names of two locations) and returns a `float` which is the distance from the first location to the second, or -1.0 if the distance is not stored in the distance map. The doctest examples in class `SpaceFleet` (`a2_space_bikes`) depend on there also being a method called `add_distance`. Make sure that it exists and is consistent with those doctests.

The choice of data structure is up to you. Make sure to define representation invariants that document any important facts about your data structure.

You may find it helpful to use a default parameter somewhere in this class. Here is an example of how they work. This function takes two parameters, but if the caller sends only one argument, it uses the default value 1 for the second argument.

```
def increase_items(lst: List[int], n: int = 1) -> None:
    """Mutate lst by adding n to each item.

    >>> grades = [80, 76, 88]
    >>> increase_items(grades, 5)
    >>> grades == [85, 81, 93]
    True
    >>> increase_items(grades)
    >>> grades == [86, 82, 94]
    True
    """
    for i in range(len(lst)):
        lst[i] += n
```

Task 2: Defining Enterprise Entities

Task 2 is completed in `a2_space_bikes.py`.

You will now define the entities that exist in your enterprise to scaffold the automated system you plan to implement.

Note that class `SpaceFleet` has its interface designed for you. You will need to implement `SpaceFleet` according to its designed interface. Carefully read the doctests of `SpaceFleet`, as they dictate what services `SpaceBike` and `Passenger` will need to provide to the client `SpaceFleet`.

You will need to design and define the classes `SpaceBike` and `Passenger` (using the Class Design Recipe) such that they provide the services required by `SpaceFleet`. Additionally, the `SpaceBike` class should allow a parameter that sets a starting location, defaulting to the constant `DEFAULT_STARTING_LOC` provided.

It is recommended that you start by implementing classes `SpaceBike` and `Passenger`, focusing on the data you know it must store and any operations that you are certain it must provide. As you start to use these classes in later steps, you will likely add new operations or make other changes. This is appropriate and a natural part of the design process. In particular, you will need to refer to task 4 for specifications on how to board Passengers and modify the SpaceBike's route. Once you have classes `SpaceBike` and `Passenger` completed and tested, you can move on to implement class `SpaceFleet`.

You may wish to refer to the

- Fleet Data file, as this is the data that your `SpaceFleet` and `SpaceBike` classes will represent
- Passenger Data file, as this is the data that your `Passenger` class will represent

Task 3: Priority Queue

Task 3 is completed in `a2_container.py`.

With your automated Passenger scheduling system, it is reasonable to anticipate that you might want to board Passengers, for example, starting with the highest bidder. This means that each Passenger would be assigned a *priority* based on their bid amount. To keep track of this priority, you will be implementing a **priority queue**.

The **priority queue** ADT is an extension of the **queue** ADT. It is a container that supports add and remove operations, with the added structure of maintaining an order with respect to a dictated priority. A priority queue always removes the item with the highest *priority*. If there is a tie for highest priority, it chooses among the tied items the one that was inserted first. (See the docstrings in class `PriorityQueue` for examples.)

... but what if you wanted to compare two ways of prioritizing Passengers, to see which yields a higher profit?

We will enable our priority queue to prioritize its entries in different ways. Our `PriorityQueue` class lets client code define what the priority is to be by passing to the initializer a function that can compare two items, annotated with the `Callable` annotation. See section 1.4 of the lecture notes.

File `a2_container.py` contains the general `Container` class, as well as a partially-complete `PriorityQueue` class. `PriorityQueue` is a child class of `Container`.

The `PriorityQueue` is implemented as a linked list, using the `_QueueNode` class. `_QueueNode` has been implemented for you.

You must complete the `PriorityQueue` methods `is_empty`, `add`, and `remove` according to the docstrings.

Task 4: Scheduling Passengers

Task 4 is completed in `a2_fleet_scheduler.py`.

The abstract class `FleetScheduler` is defined for you. You must define and implement two child classes of `FleetScheduler`, `BogoScheduler` and `GreedyScheduler`. Review the docstring of the `schedule` method in `FleetScheduler` carefully.

A Passenger is only boarded onto a SpaceBike if the **Passenger source is already in the SpaceBike's route**. Additionally, the Passenger is only permitted to board if either the SpaceBike is already travelling *from* the Passenger source *to* the destination, or the SpaceBike has enough fuel to travel from the last location in its route to the Passenger destination.

For example, if Passenger A is travelling from location X to location Y, a SpaceBike that has the following route:

(location Y -> location X)

is **not** travelling from the Passenger source to the destination. If we wanted to board Passenger A on this SpaceBike, we would need to make sure the SpaceBike has enough fuel to travel the following route:

(location Y -> location X -> location Y)

If you add any attributes to your child classes, make them private. You are permitted to define an initializer for your child classes that has a different parameter list than the one it inherits. Do **not** change the interface of any other methods in the starter code.

You **MUST** not mutate the lists passed to the `schedule` method. For example, do **NOT** call, e.g., `passengers.sort()`. You are permitted (and required) to modify the objects within the lists (e.g. adding a Passenger onboard a SpaceBike).

BogoScheduler

Named after the notoriously inefficient Bogosort, this scheduler randomly assigns Passengers to SpaceBikes in its `schedule` method.

The random algorithm you will implement is as follows:

1. Choose the next Passenger you will board randomly.
2. Attempt to board the Passenger on each SpaceBike in the SpaceFleet until you succeed. No particular order of candidate SpaceBikes is enforced. Make sure to follow the boarding requirements described above. If you are unable to board the Passenger on any bike, the Passenger is not boarded.

This scheduler is non-deterministic; i.e., if you run it multiple times on the same scheduling problem, it will produce different results.

Repeat steps (1)-(2) until all Passengers have been considered for boarding.

GreedyScheduler

The `GreedyScheduler` picks the *best* option among all the available options at any given step to implement its `schedule` method. We will determine *best* using assigned *priorities*. You will use your `PriorityQueue` and `DistanceMap` implementations for this task. You may find it helpful to define three functions for Passenger priority, and one function for SpaceBike priority. Define these functions at the module level (outside of any class) and name each with a leading underscore to indicate that it is private. Your greedy scheduler must allow three ways of prioritizing Passengers:

- Non-descending order of (distance between Passenger source and destination)
- Non-ascending order of fare bid
- Non-ascending order of (fare bid, divided by distance between Passenger source and destination)

The distance between Passenger source and destination should be retrieved from a common instance of a `DistanceMap` for all Passengers, as per the `distance` method applied to the Passenger source and destination. Each instance of the `GreedyScheduler` will only use one way of Passenger prioritization (i.e. the prioritization of Passengers will not change for the duration of the scheduler's existence; the different prioritizations are meant to be used in separate scheduling instances)

Implement the GreedyScheduler by repeating steps (1)-(4) until all Passengers have been considered for boarding, as follows:

1. Choose the next Passenger you will board according to the defined Passenger priority.
2. Choose a candidate SpaceBike to attempt to board the Passenger. The candidate SpaceBike is chosen from SpaceBikes with non-zero capacity as follows:
   - In the order of non-decreasing capacity, select the first bike that is already travelling from the Passenger source to the Passenger destination.
   - If no bikes fulfill the previous criteria, select the SpaceBike that requires the minimum additional fuel expended to travel to the Passenger's destination. The additional fuel expended is defined as the amount of fuel required for the bike to travel from the **last location in its route** to the Passenger's destination.
   - In the case of a tie for fuel expended, choose the SpaceBike with the least available capacity. In the case of a further tie for available capacity, choose the SpaceBike with the smaller ID.
   - If no bikes meet the selection criteria, then there is no candidate SpaceBike to board the current Passenger.
3. If there is no candidate SpaceBike to admit the Passenger, the Passenger is not boarded on any SpaceBike.
4. If there is a candidate SpaceBike to admit the Passenger, board the Passenger on the SpaceBike. If the SpaceBike is not yet travelling **from** the source to the destination in its route, **add the Passenger's destination to the end of the SpaceBike's route**.

This scheduler is deterministic; i.e., if you run it multiple times on the same scheduling problem, it should produce the same result.

Task 5: Command Central

You are now ready to pass along your code base to command central, so that they can run the Fleet Schedulers you have created and decide on the best path forward.

The `CommandCentral` class and its `run` method are defined for you. You will need to complete the `__init__`, `_compute_stats`, and `_print_stats` methods.

The parameter for the `init` method is a configuration dictionary, with the following possible keys (`str`) and corresponding values (`Union[str, int]`):

- `"scheduler_type"`: `"bogo"` or `"greedy"`
- `"verbosity"`: `0`, or any other non-zero integer you have chosen when implementing your `FleetScheduler` child classes
- `"passenger_priority"`: `"travel_dist"`, `"fare_bid"`, or `"fare_per_dist"`; corresponding to the different Passenger priorities as described in Task 4, the greedy scheduler subsection. If the `"scheduler_type"` is `"bogo"`, `"Passenger_priority"` does not need to be set.
- `"passenger_fname"`: a file name of a file containing data as per the passenger data file structure.
- `"galaxy_fname"`: a file name of a file containing data as per the galaxy data file structure.
- `"fleet_fname"`: a file name of a file containing data as per the fleet data file structure.

To complete `_compute_stats`, you should populate all the values for the following keys in the `<self>._stats` dictionary as follows:

- `'num_bikes'`: The total number of SpaceBikes in the SpaceFleet.
- `'num_empty_bikes'`: The total number of *empty* SpaceBikes in the SpaceFleet; i.e. SpaceBikes that have *no* Passengers assigned.
- `'average_fill_percent'`: The result of the`SpaceFleet` method `average_fill_percent`
- `'average_distance_travelled'`: The result of the `SpaceFleet` method `average_distance_travelled`
- `'vacant_seats'`: The result of the `SpaceFleet` method `vacant_seats`
- `'total_fare_collected'`: The result of the `SpaceFleet` method `total_fare_collected`
- `'deployment_cost'`: The result of the `SpaceFleet` method `total_deployment_cost`
- `'profit'`: the result of `<total_fare_collected> - <total_deployment_cost>`

File Structure

You can find files that have been prepared for the express purpose of testing your work on in the `data/testing` directory. The `data/full_passenger` and `data/full_space_fleet` directories contain data that will be used in `brute_force.py`; you are free to use these files for your own testing as well. The recommended testing order for files in the `data/testing` directory are described here.

For all files, the following will hold: There are *no* blank lines separating content in the file. Once you read a blank line in the file, you may assume the file has ended. Blank lines are lines that are *only comprised of whitespace*. Any string `s` such that Python's `str.isspace(s)` evaluates to`True` is considered to be whitespace.

The functions that read from data files all have strong preconditions allowing the code to assume inputs are valid and exactly as described in the handout.

Notice that there is little to no focus on validating data.

Galaxy Data

The galaxy data is stored in the `data` directory. It is stored as tab-separated data, where each row is guaranteed to have 3 columns. Each column is separated by one or more tab characters, and may have excess leading and trailing whitespace that must be removed. Tabs are represented in Python as `\t`.

- The first column represents the `<source>`
- The second column represents the `<destination>`
- The third column represents the distance required to travel from the `<source>` to the `<destination>`. Once stripped of whitespace, this is guaranteed to be a valid floating point value.

For example, the following string read in as a line of the file:

```
A\tB\t2.0\n
```

means that to travel from location `A` to location `B`, is a distance of `2.0`.

Note that the distance required to travel from e.g. Earth to Mars *may not be the same* as the distance required to travel from Mars to Earth. In the galaxy data file provided, you are guaranteed one of the following:

- If the distance travel from (`<a>` to `<b>`) is the same as (`<b>` to `<a>`), there will only be **one row in the entire galaxy data file** representing this.
- If the distance to travel from (`<a>` to `<b>`) is not the same as (`<b>` to `<a>`), the data will be reported in consecutive lines (i.e. if a row reports the distance to travel from `<a>` to `<b>`, the row immediately following it will report the distance to travel from `<b>` to `<a>`).

This file is guaranteed to have one or more lines.

Fleet Data

The test SpaceFleet data is found in the `data/testing` directory. There are three SpaceFleet data files:

- `space_fleet_data.txt` is a small file of SpaceBikes. You should test on this file to start with.
- `space_fleet_data_larger.txt` is a moderately sized file of SpaceBikes. You should *not* test on this file until you are sure your code works on `space_fleet_data.txt`
- `space_fleet_data_huge.txt` is a large file of SpaceBikes. You should *not* test on this file until you are sure your code works on `space_fleet_data.txt`

This file starts with a line indicating the *type* of SpaceBike (one of `Atom Bike`, `Nova Bike`, or `MC^2 Bike`).

The next **line** consists of tab-separated values, that may be separated by one or more tab characters, and may have excess leading and trailing whitespace that must be removed. The tab-separated values appear in the order as follows:

- `ID_<bike_id>`, where `<bike_id>` is a valid integer value.
- A valid integer value representing the maximum Passenger capacity of the SpaceBike.
- A valid floating point value representing the maximum fuel capacity of the SpaceBike in ML.
- A valid floating point value representing the fuel usage rate of the SpaceBike in ML/AU.

The next line, if there is one present, repeats the same two-line pattern as above, representing the next SpaceBike in the Fleet.

Every SpaceBike in the Fleet has a starting position of the `DEFAULT_STARTING_LOC` constant in `a2_space_bikes.py`. You should make use of this constant, as we may change it to ensure your code still works when the default starting position is changed.

This file is guaranteed to have two or more lines.

Passenger Data

The test Passenger data is found in the `data/testing` directory. There are three Passenger data files:

- `passenger_off_peak.txt` is a small file of 35 Passengers. You should test on this file to start with.
- `passenger_on_peak.txt` is a file of 800 Passengers. You should *not* test on this file until you are sure your code works on `passenger_off_peak.txt`
- `passenger_huge.txt` is a file of 50,000 Passengers. You should *not* test on this file until you are sure your code works on `passenger_on_peak.txt`

The format of any Passenger files adheres to the following format:

The first line of the file is the name of a Passenger. The next **line** consists of tab-separated values, that may be separated by one or more tab characters, and may have excess leading and trailing whitespace that must be removed. The tab-separated values appear in the order as follows:

- `BID:` followed by one or more spaces, followed by a valid floating point value representing the potential Passenger's bid amount (in millions of dollars).
- `SOURCE:` followed by one or more spaces, followed by the source location of the Passenger.
- `DEST:` followed by one or more spaces, followed by the destination location of the Passenger.

The next line, if there is one present, repeats the same two-line pattern as above, representing the next Passenger.

This file is guaranteed to have two or more lines.

Submission Instructions

Submit `a2_container.py`, `a2_distance_map.py`, `a2_space_bikes.py`, `a2_command_central.py`, and `a2_fleet_scheduler.py` on MarkUs. We strongly recommend that you submit early and often. We will grade the latest version you submit. To avoid accidentally incurring late penalties, do not submit files after the time you intend to submit. Run the tests on MarkUs one last time before the due date to make sure that you didn't accidentally submit the wrong file(s)!

Grading Scheme

There will be no marks associated with defining your own test cases with pytest or hypothesis. The grading scheme is as follows:

- pyTA: 10 marks, with 1 mark deducted for each occurrence of each pyTA error.
- following the coding guidelines: 5 marks
- self-tests, provided in MarkUs: 20 marks
- `Passenger` and `SpaceBike` classes: will not be tested directly, other than for aspects that are proscribed by the doctest examples in class `SpaceFleet`
- hidden tests: 65 marks

Late Policy

- 0% deduction for the first hour
- then 5% deduction per hour for the next 5 hours
- then 15% deduction for the next 5 hours
- after 11 hours, no lates are accepted.

Appendix A: Data Sources & Attribution

Data was sourced from:

- Distances of galactic bodies: http://jpl.nasa.gov/edu/pdfs/scaless_reference.pdf
- Fake names for Passengers: https://pypi.org/project/names/

This assignment was adapted from the Winter 2021 offering of CSC148 (A1 created by Diane Horton, Ian Berlott-Attwell, Jonathan Calver, Sophia Huynh, Maryam Majedi, and Jaisie Sin). The content was adapted for the Summer 2021 offering by Saima Ali and Marina Tawfik.

A heartfelt thanks to Ian Berlott-Attwell and Sophia Huynh for their continued efforts in the creation and delivery of this assessment!

Appendix B: Example of Greedy Scheduling Algorithm

Galaxy Distances

Here, we will assume that the distance from `<a>` to `<b>` is the same as the distance from `<b>` to `<a>` to simplify the example. In general, this is **not** a valid assumption for this assignment.

Source	Destination	Distance (AU)
Earth	Venus	4.5
Sun	Venus	1.0
Earth	Mars	2.0
Venus	Mars	4.0

Passengers

In this example, we will prioritize Passengers based on their fare bid.

Passenger	Fare Bid (M$)	Source	Destination
Laura Engelman	2.5	Earth	Venus
Jonathan McMyers	1.5	Sun	Venus
MaryAnn Jacobs	1.0	Earth	Mars
Robert Robarts	0.5	Venus	Sun
Lily Jenkins	0.2	Venus	Sun
Melanie Kim	0.1	Earth	Venus

SpaceFleet

Note that we **always** prioritize the SpaceBikes in non-descending order of available capacity. The default starting location for every spacebike is set to Earth (constant defined in `a2_space_bikes.py`)

SpaceBike ID	Type	Max Passengers	Fuel Capacity (ML)	Fuel Usage Rate (ML/AU)	Route
1	Atom Bike	2	4.0	0.5	Earth
2	Atom Bike	4	4.0	0.5	Earth
3	Nova Bike	20	8.0	1.0	Earth

Iteration 1

Passenger with the highest bid: Laura Engelman

Check each bike in the order of non-descending available capacity.

SpaceBike 1 (capacity 2):

- Is Passenger source (Earth) in SpaceBike 1's current route?
  - Yes.
- Does SpaceBike 1's current route travel *from* source (Earth) *to* destination (Venus)?
  - No.
- Does SpaceBike 1 have enough fuel to add destination (Venus) to the end of its route?
  - Yes.
    * Proposed new route: Earth -> Venus
    * Distance: 4.5
    * Fuel required: 4.5 * 0.5 = 2.25 ML
    * Fuel available: 4.0 ML

Board Laura Engelman on SpaceBike 1.

Updated SpaceBikes

- SpaceBike 1
  - Available Capacity: 1
    * Boarded Passengers: Laura Engelman
  - Route: Earth -> Venus

Iteration 2

Passenger with the highest bid: Jonathan McMyers

Check each bike in the order of non-descending available capacity.

SpaceBike 1 (capacity 1):

- Is Passenger source (Sun) in SpaceBike 1's current route?
  - No.
- Check next SpaceBike.

SpaceBike 2 (capacity 4):

- Is Passenger source (Sun) in SpaceBike 2's current route?
  - No.
- Check next SpaceBike.

SpaceBike 3 (capacity 20):

- Is Passenger source (Sun) in SpaceBike 3's current route?
  - No.

No more SpaceBikes to check. Passenger not boarded.

Iteration 3

Passenger with the highest bid: MaryAnn Jacobs

Check each bike in the order of non-descending available capacity.

SpaceBike 1 (capacity 1):

- Is Passenger source (Earth) in SpaceBike 1's current route?
  - Yes.
- Does SpaceBike 1's current route travel *from* source (Earth) *to* destination (Mars)?
  - No.
- Does SpaceBike 1 have enough fuel to add destination (Mars) to the end of its route?
  - No.
    - Proposed new route: Earth -> Venus -> Mars
    - Distance: 4.5 + 4.0 = 8.5
    - Fuel required: 8.5 * 0.5 = 4.25 ML
    - Fuel available: 4.0 ML

Check next SpaceBike.

SpaceBike 2 (capacity 4):

- Is Passenger source (Earth) in SpaceBike 1's current route?
  - Yes.
- Does SpaceBike 2's current route travel *from* source (Earth) *to* destination (Mars)?
  - No.
- Does SpaceBike 2 have enough fuel to add destination (Mars) to the end of its route?
  - Yes.
    - Proposed new route: Earth -> Mars
    - Distance: 2.0
    - Fuel required: 2.0 * 0.5 = 1.0 ML
    - Fuel available: 4.0 ML

Board MaryAnn Jacobs on SpaceBike 2.

Updated SpaceBikes

- SpaceBike 2
  - Available Capacity: 3
    - Boarded Passengers: MaryAnn Jacobs
  - Route: Earth -> Mars

Iteration 4

Passenger with the highest bid: Robert Robarts

Check each bike in the order of non-descending available capacity.

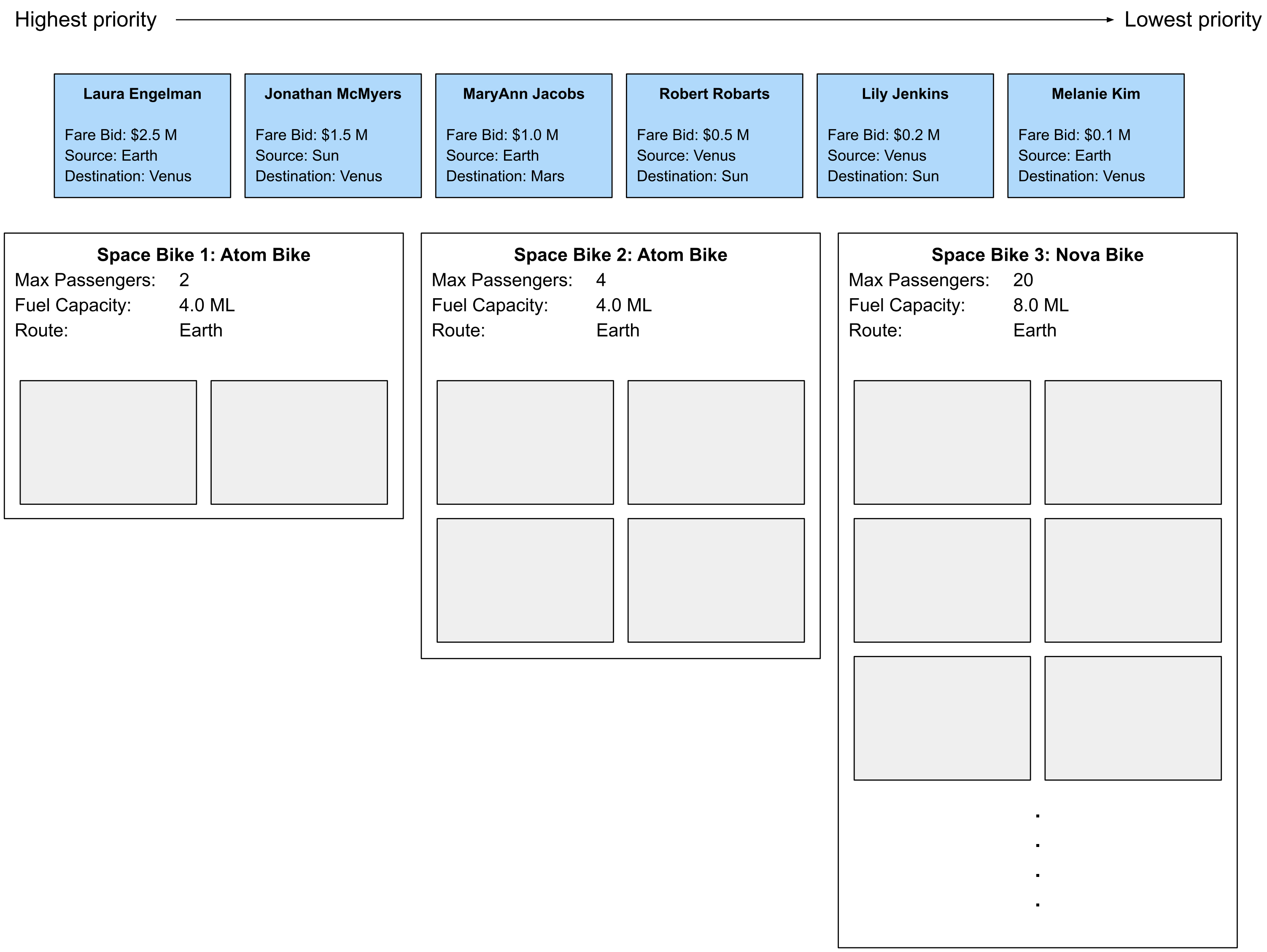
SpaceBike 1 (capacity 1):

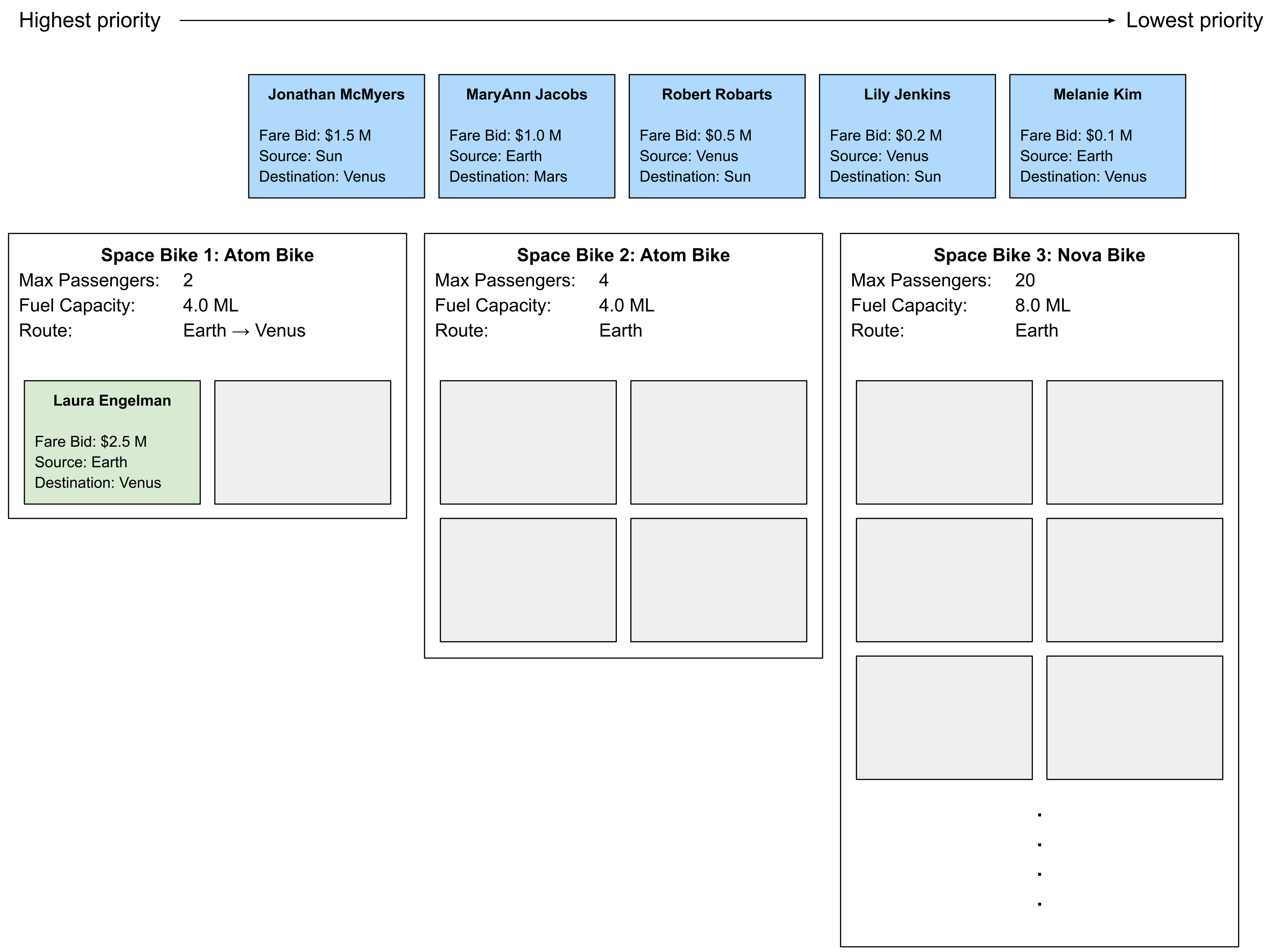
- Is Passenger source (Venus) in SpaceBike 1's current route?
  - Yes.
- Does SpaceBike 1's current route travel *from* source (Venus) *to* destination (Sun)?
  - No.
- Does SpaceBike 1 have enough fuel to add destination (Sun) to the end of its route?
  - Yes.
    - Proposed new route: Earth -> Venus -> Sun
    - Distance: 4.5 + 1.0 = 5.5
    - Fuel required: 5.5 * 0.5 = 2.75 ML
    - Fuel available: 4.0 ML
- Board Robert Robarts on SpaceBike 1.

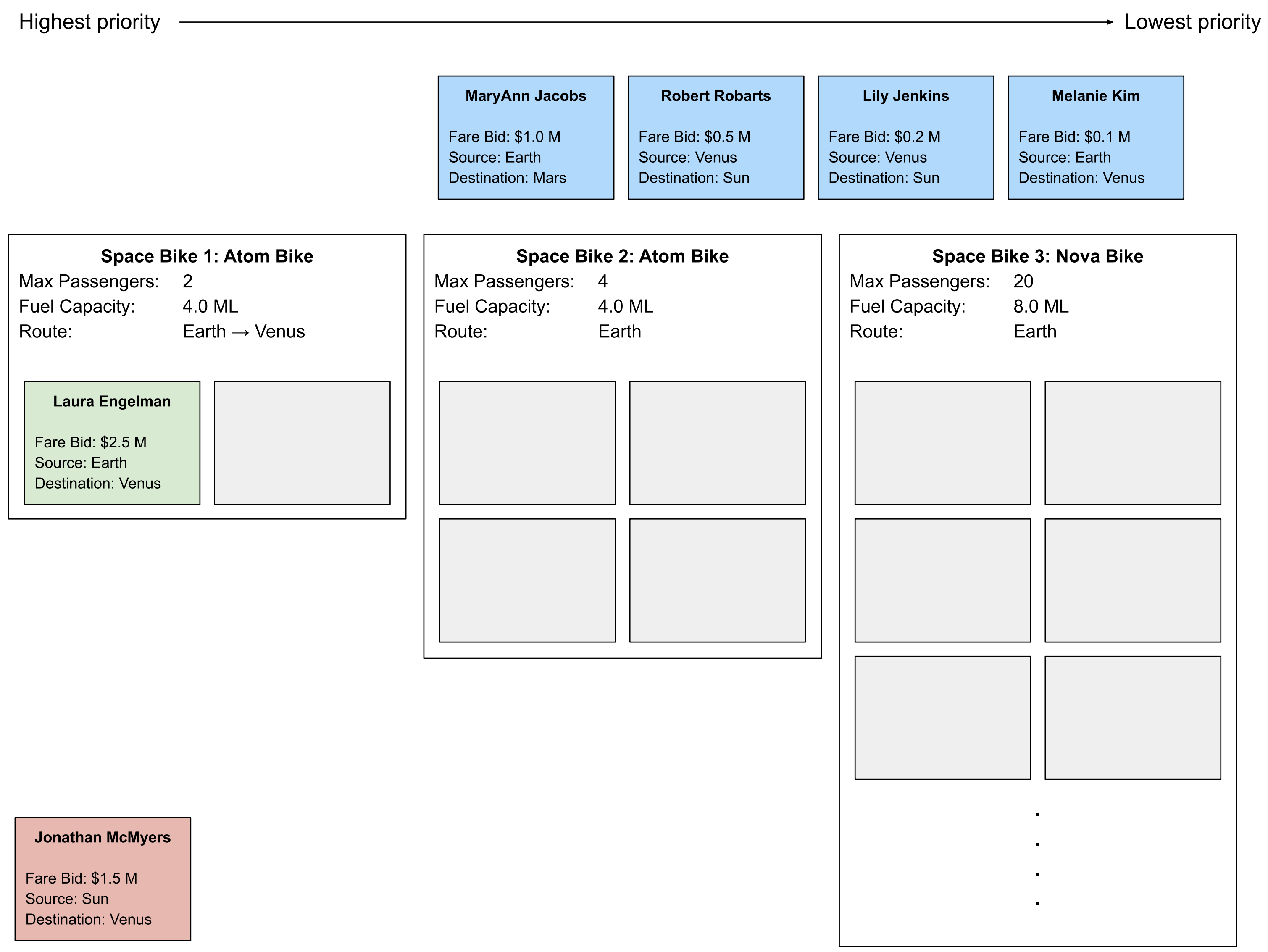
Updated SpaceBikes

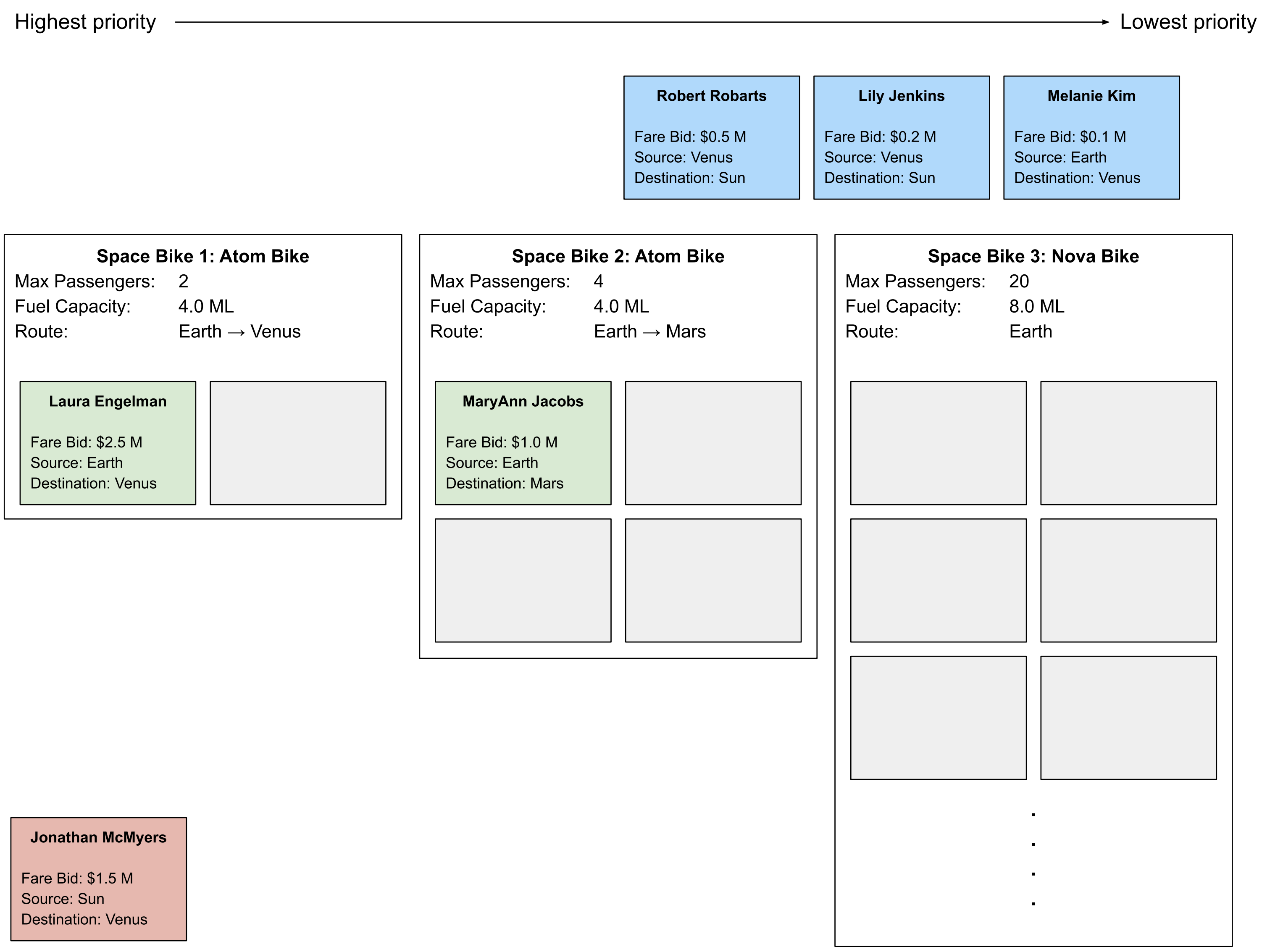
- SpaceBike 1
  - Available Capacity: 0
    - Boarded Passengers: Laura Engelman, Robert Robarts
  - Route: Earth -> Venus -> Sun

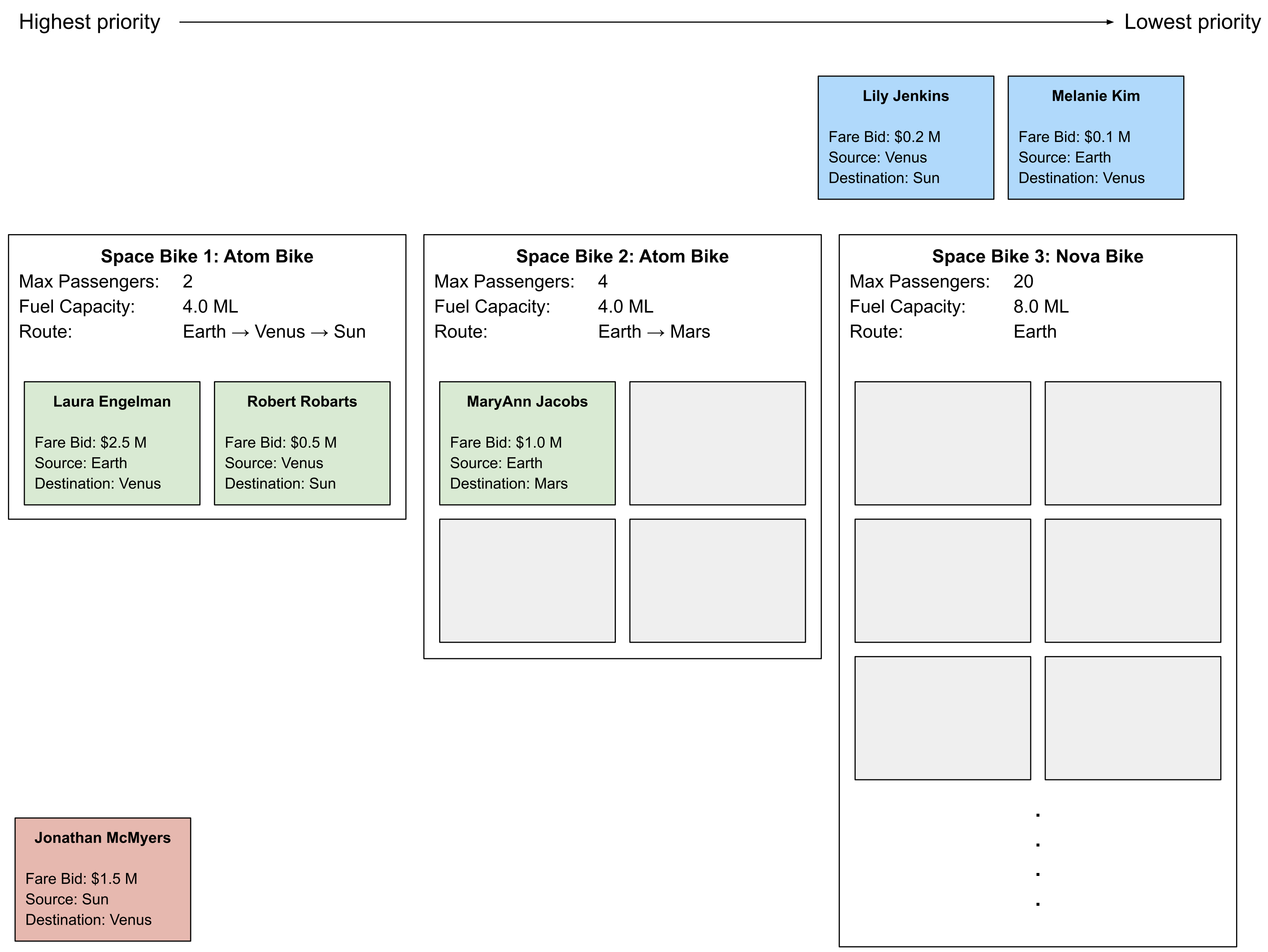
Visualization




Highest priority
Lowest priority
Jonathan McMyers
Fare Bid: $1.5 M
Source: Sun
Destination: Venus
MaryAnn Jacobs
Fare Bid: $1.0 M
Source: Earth
Destination: Mars
Robert Robarts
Fare Bid: $0.5 M
Source: Venus
Destination: Sun
Lily Jenkins
Fare Bid: $0.2 M
Source: Venus
Destination: Sun
Melanie Kim
Fare Bid: $0.1 M
Source: Earth
Destination: Venus
Space Bike 1: Atom Bike
Max Passengers: 2
Fuel Capacity: 4.0 ML
Route: Earth → Venus
Laura Engelman
Fare Bid: $2.5 M
Source: Earth
Destination: Venus
Space Bike 2: Atom Bike
Max Passengers: 4
Fuel Capacity: 4.0 ML
Route: Earth
Space Bike 3: Nova Bike
Max Passengers: 20
Fuel Capacity: 8.0 ML
Route: Earth


Highest priority
Lowest priority
MaryAnn Jacobs
Fare Bid: $1.0 M
Source: Earth
Destination: Mars
Robert Robarts
Fare Bid: $0.5 M
Source: Venus
Destination: Sun
Lily Jenkins
Fare Bid: $0.2 M
Source: Venus
Destination: Sun
Melanie Kim
Fare Bid: $0.1 M
Source: Earth
Destination: Venus
Space Bike 1: Atom Bike
Max Passengers: 2
Fuel Capacity: 4.0 ML
Route: Earth → Venus
Laura Engelman
Fare Bid: $2.5 M
Source: Earth
Destination: Venus
Space Bike 2: Atom Bike
Max Passengers: 4
Fuel Capacity: 4.0 ML
Route: Earth
Space Bike 3: Nova Bike
Max Passengers: 20
Fuel Capacity: 8.0 ML
Route: Earth
Jonathan McMyers
Fare Bid: $1.5 M
Source: Sun
Destination: Venus


Highest priority
Lowest priority
Robert Robarts
Fare Bid: $0.5 M
Source: Venus
Destination: Sun
Lily Jenkins
Fare Bid: $0.2 M
Source: Venus
Destination: Sun
Melanie Kim
Fare Bid: $0.1 M
Source: Earth
Destination: Venus
Space Bike 1: Atom Bike
Max Passengers: 2
Fuel Capacity: 4.0 ML
Route: Earth → Venus
Laura Engelman
Fare Bid: $2.5 M
Source: Earth
Destination: Venus
Space Bike 2: Atom Bike
Max Passengers: 4
Fuel Capacity: 4.0 ML
Route: Earth → Mars
MaryAnn Jacobs
Fare Bid: $1.0 M
Source: Earth
Destination: Mars
Space Bike 3: Nova Bike
Max Passengers: 20
Fuel Capacity: 8.0 ML
Route: Earth
Jonathan McMyers
Fare Bid: $1.5 M
Source: Sun
Destination: Venus


Highest priority
Lowest priority
Lily Jenkins
Fare Bid: $0.2 M
Source: Venus
Destination: Sun
Melanie Kim
Fare Bid: $0.1 M
Source: Earth
Destination: Venus
Space Bike 1: Atom Bike
Max Passengers: 2
Fuel Capacity: 4.0 ML
Route: Earth → Venus → Sun
Laura Engelman
Fare Bid: $2.5 M
Source: Earth
Destination: Venus
Robert Robarts
Fare Bid: $0.5 M
Source: Venus
Destination: Sun
Space Bike 2: Atom Bike
Max Passengers: 4
Fuel Capacity: 4.0 ML
Route: Earth → Mars
MaryAnn Jacobs
Fare Bid: $1.0 M
Source: Earth
Destination: Mars
Space Bike 3: Nova Bike
Max Passengers: 20
Fuel Capacity: 8.0 ML
Route: Earth
Jonathan McMyers
Fare Bid: $1.5 M
Source: Sun
Destination: Venus