**Simulator Tutorial**

Trever Hallock

Tamlyn Harley

Matthew Stanley

Cody Zhang

CONTENTS

Purpose …………………………………………………………………………………………... 2

Getting Started …………………………………………………………………………………... 3

What Goes In ……………………………………………………………………………………. 4

*Cities …………………………………………………………………………………………………………. 4*

*Actions ……………………………………………………………………………………………………….. 5*

*Solutions ……………………………………………………………………………………………………... 6*

*Viability ……………………………………………………………………………………………………… 7*

*Translation ………………………………………………………………………………………………….. 7*

What Comes Out ………………………………………………………………………………… 8

Example ………………………………………………………………………………………..... 9

*Objects and the Dot Operator …………………………………………………………………………... 13*

Conclusion ……………………………………………………………………………………... 14

PURPOSE

The purpose of our simulator is to generate a random city / route to test for Sam’s Hauling, as well as provide a function that can take a solution and demonstrate its efficiency based on various metrics. The file is written using MATLAB, and this manual is intended to show how to use this program and deal with its output.

This program is a work in progress, and the team is eager to receive feedback! If anyone has any suggestions for improvement, or spots any errors, please feel free to either post this feedback to our Canvas discussion board, where we will work diligently to implement all of the changes needed or requested.

GETTING STARTED

To get started, one must simply go to our Canvas discussion page and download the .zip file that is located there. (This is constantly being updated, so make sure to get the latest version if changes are implemented). Then, if one does not have MATLAB installed on their computer, the computer lab has it on the workstations; additionally, MATLAB is sometimes offered free to students through the university.

Once downloaded, you will need to change the directory on MATLAB to reflect …\animated-dubstep-master\matlab, where our files are stored. This can be done through the MATLAB function 'cd' or by browsing through the current path on the left-hand panel of MATLAB.

Once in the correct directory, you should see several subfolders and a file called env.m. You must run this file for all of the subdirectories to be enabled, and this should be your first step; so, from the MATLAB command window, type “env” and it will enable everything for you. Thus, there are three easy steps:

1. **Download animated-dubstep-master.zip from Canvas, and unzip the files**
2. **Change the MATLAB path to reflect …\animated-dubstep-master\matlab**
3. **Run the “env” script in order to set the matlab environment**

This is all you need to get started simulating!

WHAT GOES IN

Now that you’ve installed the simulator, how does it work? Well, there are two main components to the simulator: cities and solutions.

What are cities?

Cities are objects (if you don’t know this programming term, don’t worry, it isn’t necessary to understand) that store multiple pieces of information. The city stores all of the data about routes, stops, customer requests, landfills, staging areas, etc. It is an entire set of data that makes up the problem statement. The function generate\_city(R, L, Y, D) generates a city at random, based on these arguments:

**R is the number of customer requests to be generated**

**L is the number of landfills to be generated**

**Y is the number of staging areas, or yards, to be generated**

**D is the number of drivers that work in that city**

This function creates a random set of customer requests, landfills, staging areas, trucks, etc. for us to run a simulation on. Everything is random, so that you can create a diverse set cities to run simulations through. To get a visual overview of the city, simply use the display\_city function. So, in order to create a city and see what it looks like, follow these steps:

1. **From the MATLAB command window, type in c = generate\_city(R, L, Y, D) where you replace the arguments with numbers of your own**
2. **From the MATLAB command window, type display\_city(c)**

The visual output is not at all necessary to create a city, but it is helpful to understanding what was generated. *For those who understand object oriented programming, they can certainly create a city by hand manually, but there is a lot of data to be filled in.*

What are actions?

A city contains actions – also called stops – and the first thing anyone will notice with our simulator is that there are a *lot* of actions for any city. Why are there so many, and what are they?

An action is simply something a driver can do; that’s it. An action has seven main parts to it:

**The operation –** this can be P (pickup), D (dropoff), R (replace), S (stage), U (unstage), or E (empty)

**The in-size** – this is the size of the dumpster that is being brought to the action or stop, as a numerical value between of 0, 6, 9, 12, or 16. The 0 represents no dumpster and the others represent the four sizes of containers we are dealing with

**The out-size** – this is the same thing as in-size, but it is what size of dumpster the driver is supposed to leave with

**The start**-**time** – this is the time, in seconds, when this action can start being performed; for example, if the start-time is 10,000, this means that 10,000 seconds must elapse during the simulation before this action can be performed

**The stop-time** – this is the time, in seconds, when an action must be performed by; together with start-time, these model time window constraints

**The wait**-**time** – this is the amount of time it takes to actually complete the action (note that this is different from the travel time it takes to get to the location, which is modeled in a different place)

**The location –** this is where the action is actually located; it is an index into the array of locations that we have, telling us which location has this particular action. It is also an index into the matrix of distances and durations between locations.

Different areas have different kinds of actions:

**Staging areas** have 8 actions each, as shown in this table:

|  |  |  |  |
| --- | --- | --- | --- |
| STAGING AREA | Operation | In-size | Out-size |
| ACTION 1 | STAGE | 1 | 0 |
| ACTION 2 | STAGE | 2 | 0 |
| ACTION 3 | STAGE | 3 | 0 |
| ACTION 4 | STAGE | 4 | 0 |
| ACTION 5 | UNSTAGE | 0 | 1 |
| ACTION 6 | UNSTAGE | 0 | 2 |
| ACTION 7 | UNSTAGE | 0 | 3 |
| ACTION 8 | UNSTAGE | 0 | 4 |

This is because, for example, if you want to unstage (pickup) a size 3 dumpster, you must have an empty truck; thus, you would pick the seventh action, which means your in-size is 0 and your out-size is 3. *Why this is needed will become clear once a solution is given.*

Similarly, landfills have four actions each, one for each kind of dumpster you’ll be bringing there to empty. Customer requests only have one action each, as they will always have a predetermined in-size and out-size.

Now that we’ve covered cities, let’s look at solutions:

What are solutions?

A solution is a matrix; each row is a driver, and each corresponding column entry is what that driver will do in order. A negative one means that the driver does nothing (it is the end of his route). Here is an example:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| EXAMPLE SOLUTION MATRIX | | | | | | |
| 2 | 4 | 19 | 22 | -1 | -1 | -1 |
| 10 | 17 | 19 | 44 | 11 | 13 | 5 |
| 6 | 19 | -1 | -1 | -1 | -1 | -1 |

How is this interpreted? Well, as we said, each row is a driver; thus, this city has three drivers. Driver one performs action 2, then 4, then 19, then 22, and then he is done for the day. Similarly, driver three performs actions 6 and 19, in that order, then finishes his day. Driver two is the busiest, obviously, but his actions are just as easily read.

As you can see, it was necessary to encode what kinds of dumpsters the driver was dropping off, picking up, etc. with the in-size and out-size, so that this matrix would be feasible. **This encoding of the solution into a matrix is why there are so many actions associated with staging areas and landfills.** As an example, imagine that stop 19 is a staging area. Because of how the actions are structured, given any city, we will know exactly what kind of action the driver is performing at that staging area. Actions 18-25 may all be at that same staging area, but they all mean different things.

Viability

Some solutions will contain errors that the simulator will recognize as non-feasible solutions. It will return the variable feasible = false if this is the case. Many things, too many to cover, will cause it to error out; but, for example, if a driver visits a landfill followed by another landfill, this makes no sense and it will error. Or, if a driver visits a staging area to get a size 1 dumpster, but that staging area has none of those in inventory. (If you want to know what these constraints are, refer to the matlab/src directory, or the model.tex file within the doc folder.)

The simulator will continue to run to the best of its ability when dealing with a non-viable solution, but sometimes it will terminate. The errors it finds will be displayed in the main window of MATLAB.

Our suite of functions also comes with a generate\_rand\_solution function that takes a city as an argument; however, beware, this random solution is rarely viable.

Translation

This function takes the output from the user interface team (who are working with Excel) and translates it into our city structure in MATLAB. It also finds the coordinateness of the addresses, so that you can plot the city. This will be a work in progress until the UI team has finalized their output format. For example, the sample data given by the UI team is also in our repository under …/matlab/test/example\_ui\_data. If you want a city based on this data, you can simply enter the matlab command “c = translate('test/example\_ui\_data');” This city will represent the data found in this city.

WHAT COMES OUT

The simulate function takes in a city and a solution matrix as arguments, and outputs these metrics:

**Feasible** – this returns false if the solution encountered an error that makes it a non-viable solution; it returns true otherwise

**Times** – this returns, as a vector, all of the times it took each driver to complete his assigned route, based on the solution matrix given

**Distances –** this returns, as a vector, all of the distances traveled by each driver based on his assigned route

**Number Serviced** – this tells you how many of your customer requests were actually completed

**Fees** – this shows you how many fees you accrued on this route, based on the costs of landfills

**Inventories**– This lets you see how many dumpsters remain at each staging area at the end of the day

To run the simulator, simply create a city, your solution matrix, and from the main window of MATLAB call simulate(city, solution) with the variable names appropriately replaced.

EXAMPLE

Now, we’re going to give a concrete example of how to run our simulator. Then, we change our directory to …/animated-dubstep-master/matlab, and in the left panel we see a function called env.m (we’re assuming this example is running on Windows). To gain access to all of the subdirectories, we type the command:

>> env

into the MATLAB command window. Now, we will have access to all of the subdirectories needed to run the simulator.

Secondly, we generate a random city with five customer requests, three landfills, two staging areas, and two drivers, and assign it to the variable c:

>> c = generate\_city(5, 3, 2, 2)

c =

city with properties:

number\_of\_actions: 33

number\_of\_locations: 10

number\_of\_drivers: 2

number\_of\_staging\_areas: 2

number\_of\_requests: 5

number\_of\_landfills: 3

actions: [1x33 action]

yards: [1x2 staging\_area]

landfills: [1x3 landfill]

durations: [10x10 double]

distances: [10x10 double]

locs: [10x2 double]

start\_location: 1

max\_time: 43200

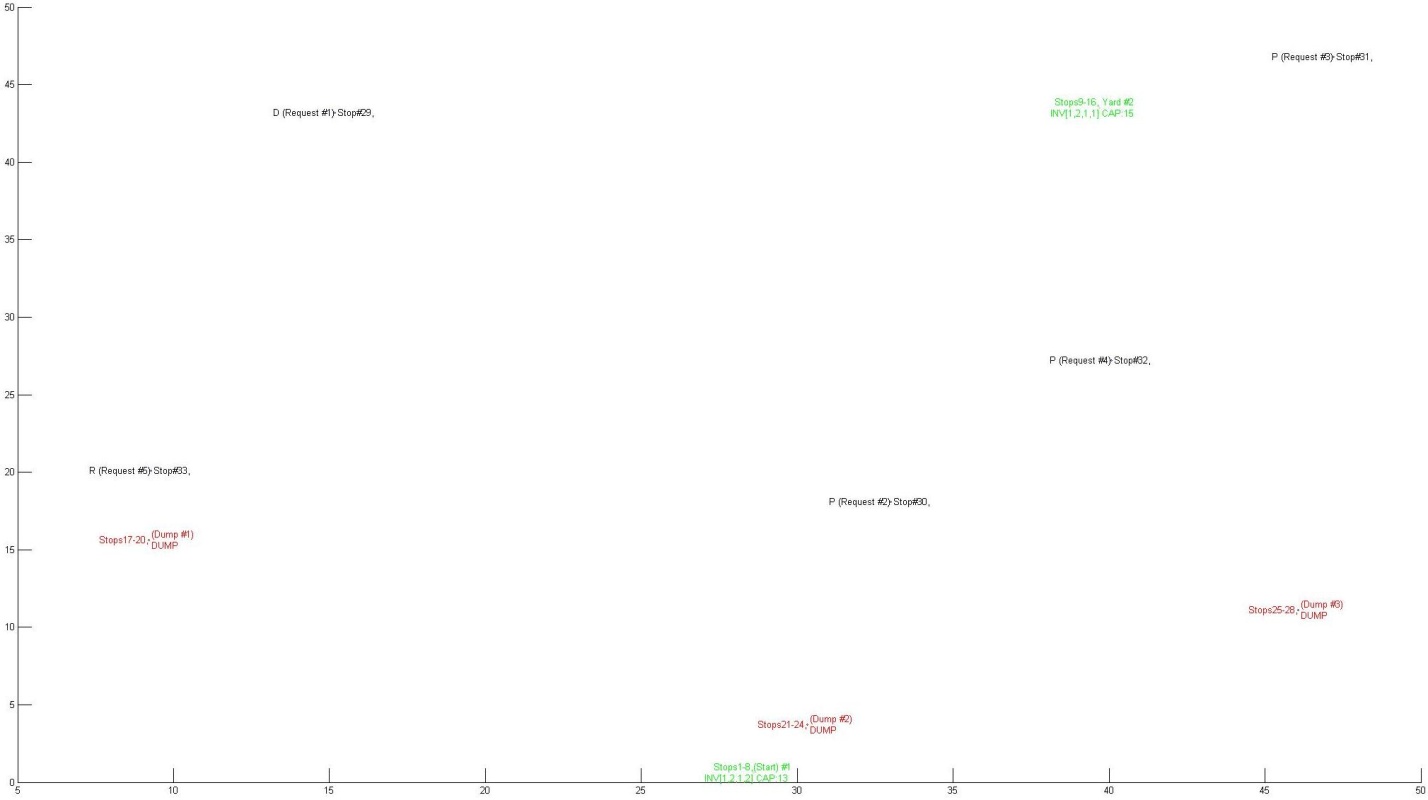
truck\_types: [1 0]

location\_to\_landfill: [0 0 1 2 3 0 0 0 0 0]

location\_to\_stagingarea: [1 2 0 0 0 0 0 0 0 0]

This displays the various attributes of our city, but don’t worry about them all yet – we will cover the necessary points as we go along. Right now, just remember that all drivers are assumed to start at location 1, and that there are 43,200 seconds in a working day (max\_time).

Next, we will run the functions display\_city on our created city, and this is the output:

>> display\_city(c)

The axes show the distances between points, but it is only 50x50 for a scale – the interpretation might vary. As you can see, the landfills, staging areas, and customer requests are all spaced out randomly.

The type of customer request given is located next to each request: P for pickup, D for dropoff, and R for replace. For the yards, remember that although they are only one location, each has eight actions associated with them, depending on what kind of dumpster we’re either bringing to them or picking up. For reference, here is the table from before listing how these stops work:

|  |  |  |  |
| --- | --- | --- | --- |
| STAGING AREA | Operation | In-size | Out-size |
| ACTION 1 | STAGE | 1 | 0 |
| ACTION 2 | STAGE | 2 | 0 |
| ACTION 3 | STAGE | 3 | 0 |
| ACTION 4 | STAGE | 4 | 0 |
| ACTION 5 | UNSTAGE | 0 | 1 |
| ACTION 6 | UNSTAGE | 0 | 2 |
| ACTION 7 | UNSTAGE | 0 | 3 |
| ACTION 8 | UNSTAGE | 0 | 4 |

So, let’s look at yard #2, which has actions 9-16 assigned to it. This means that, for example, if a driver is assigned action 14, he would be going to yard #2, picking up a size 2 dumpster (as this is the sixth action assigned to this yard). Similarly, with landfills, they each have four actions associated with them; therefore, given this city, if a driver is assigned action 18, he will be going to landfill #1 to empty a size 2 container (as this is the second action assigned to landfill #1).

The next step is to make a solution to run through the city. As explained before, a solution is just a matrix with the number of rows equal to the number of drivers. The length of the rows is arbitrary, depending on the solution given. A value of -1 in any column means that no action takes place there, and the driver is done. Therefore, let’s create a matrix that runs through this city:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ONE POSSIBLE SOLUTION MATRIX FOR THIS CITY | | | | | | |
| 30 | 25 | 9 | 31 | 28 | 29 | -1 |
| 5 | 33 | 19 | 11 | 32 | 22 | 2 |

Now, to go through each driver’s route: Driver one starts at yard #1, as do all the drivers, and he first goes to customer request #2, which corresponds to action 30. This is a pickup, and although you can’t see it from the city layout, you can type in the MATLAB command c.actions(30).out\_size to see what size of dumpster he is picking up at this location. In this case, he is picking up a size 6 dumpster (which, in our format, is option 1) – and therefore, he will need to go to a landfill and empty this size 6 dumpster. Thus, we pick action 25 which sends him to landfill #3 to empty the size 1 container. Then we send him to action 9 to stage (drop-off) that size container to yard #2, and then out to customer request 3 which is action 31. Here, he picks up a size 16 (4) dumpster, which he then empties at landfill #3 with action 28. Finally, he is sent out to drop off this size container at customer request #1 which is action 29. In our simulator, he is done, because it is assumed that he will return back to yard #1 at the end of the day – it is not necessary to input this final action.

Now, let’s examine driver 2: He first picks up a size 6 (1) dumpster from yard #1 with action 5; he then drops off this size at the replace, which is customer request #5 and action 33. He leaves with a full size 12 (3) dumpster, and so he goes to landfill #1 and empties this container using action 19. Then, he goes to yard #2 to drop off (stage) this size of dumpster using action 11, and then picks up a size 9 (2) dumpster from customer request #4 which is action 32. He empties this at landfill #2, using action 22. Finally, he returns back to yard #1 to stage (put back) this dumpster there with action 2.

Now that we have the solution, we can input it into MATLAB as a matrix, using the command:

>> sol = [30 25 9 31 28 29 -1; 5 33 19 11 32 22 2]

This command generates a matrix called sol in MATLAB. Once this is completed, we are ready to run the simulator; to do this, one can simply type the command:

>> [feasible, times, distances, num\_serviced, fees] = simulate(c, sol)

This will load all of the data output from the simulator into the variables on the left hand side, which can then be viewed from the main window by just typing in their corresponding names. It is really as simple as that!

We have included a function called generate\_solution which will create a viable solution for any city, but beware: *This function is still a work in progress, and the solutions it generates should in no way be relied upon as efficient.* We simply included it so that you, the user, can try and create random – yet viable – solutions to practice using the simulator. Also note, that this function is currently VERY slow.

There are also two optional arguments in the simulate function, but these aren’t necessary. The first is v for verbose, and if this is true, then detailed error messages are given; the second is checkall, and this is defaulted to true. However, in some rare cases we may not want to check all of the errors on a solution shown to be infeasible in order to increase processing time, so it is an option we can turn off. *As a rule of thumb, however, do not change the values of these arguments.*

Objects and the Dot Operator

This section is really only for those that do not understand the fundamentals of object oriented programming, but want to delve more deeply into the simulator. If this is you, then you will need to know a few things:

Most of the data encoded within the simulator is created using *objects.* Objects are simply collections of data. They are ways of encapsulating things together. Each object is unique, in that it has its own data. Let’s look at one of the objects we have in the simulator, which is a simple one, called staging\_area: It has three data pieces, capacity, inventory, and location. Now, supposed we have two staging areas, *one* and *two.* Each has its own **unique** capacity, inventory, and location; that is, *one*’s data is **not** the same as *two*’s – they each have their own version of the three variables listed.

In order to view, access, or modify the data, one must employ something called the *dot operator.* So, let’s say we want to view *one*’s location variable. One would simply enter the command:

>> one.location

This is because location belongs to the object called *one*, so we access it by a dot operator. Objects can also be nested inside of each other; for example, the city object has an array of other objects, called actions. Let’s say we want to know what action type city c’s action # 34 is. To do this, we simply chain together the dot operator:

>> c.actions(34).operation

Notice that there are two dots there, because we are going down two levels of objects. If you will be constructing a city, action, or anything else from scratch using code, this is necessary knowledge to have on how to access and/or modify the correct data.

CONCLUSION

I hope this gives everyone a good understanding of what our software can do; if you have any questions, feel free to contact our team through Canvas. Our team is comprised of the members:

Trever Hallock

Tamlyn Harley

Matthew Stanley

Cody Zhang

Thank you for your time, and have fun simulating! And remember – we’re here to help you!