WoW-IO - External Documentation

Oleg Kolosov, Tom Herman, Ido Zohar, Gala Yadgar
Computer Science Department, Technion

1 Introduction

The World of Warcraft Avatar History Dataset (WoWAH) is based on 3 years of gathering user information in World of Warcraft. The data is gathered in granularity of 10 minutes (we will address this unit of time as 1 *virtual-time*).

Our project creates IO streams out of this dataset, by simulating the game and user interactions.

For example, an IO request: "A_3, 2400.0, LO_k_156_171, 0" means that the Avatar with ID=3 asks to read the object LO_k_156_171 (the (156,171) location in Kalimdor) at time 2400.0 seconds from the start of this scene.

The data in WoWAH is not continuous throughout the collection period, sometimes there are "holes" of couple days (!) without data.
Therefore, in order to simulate the game continuously – we will split the dataset to continuous periods of time in which there is no lack of data, each continuous period of time will be addressed as "Scene".
Then we will try to simulate the IO requests done throughout the scenes.

2 Project Structure

Project Repository – github.com/olekol33/WoW-IO.

The structure of the project directory:

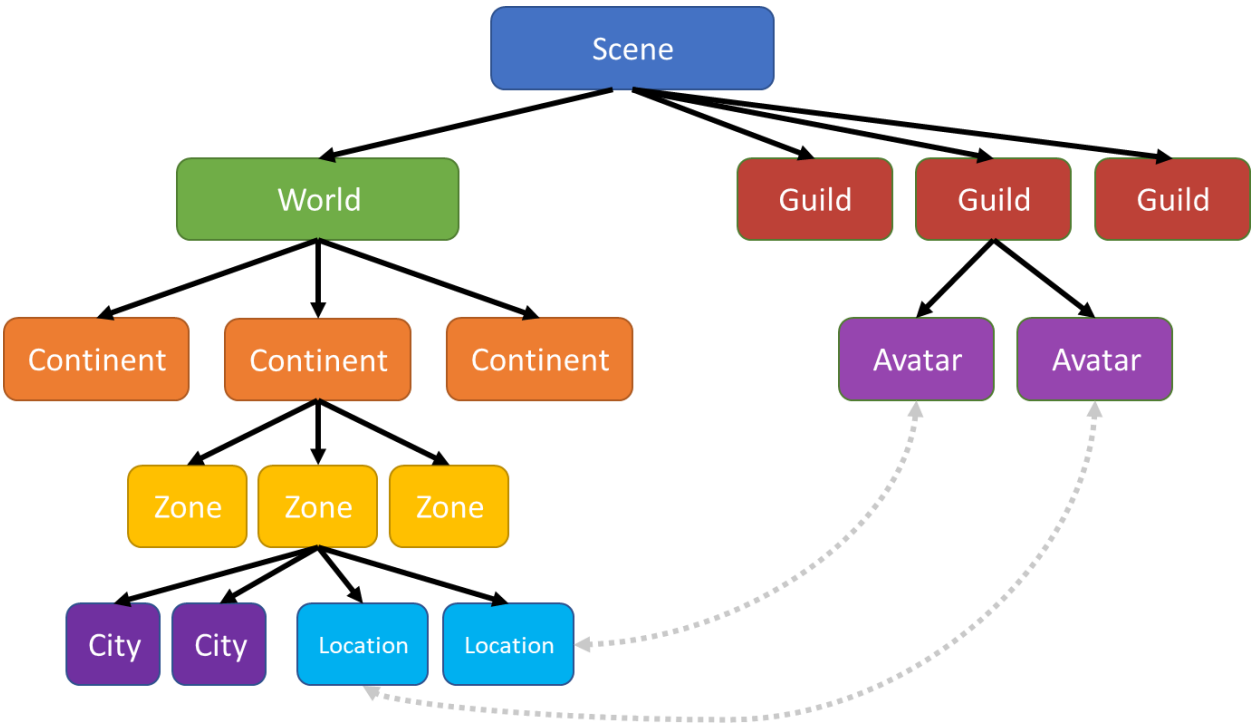
- **Modules**: The runtime-representation objects of the dataset and scenes.
- **Scripts:** The scripts for the actions available with wow.py.
- **Maps:**
    - cities.csv – describes the location and type of all the cities in the world (generated by ***`wow.py cities`***).
    - neighbors.txt – describes the neighbors-graph between zones in the world.
    - {continent}.csv – file for each continent that describes the zones in it (boundaries, number of cities per city-type).
    - {continent}.png – image for each continent that represent the zones partitions map in that continent (generated by ***`wow.py maps`***).
    - {continent}.pickle – numpy representation for each continent that represents the zones partitions map in that continent (generated by ***`wow.py maps`***).
- **Graphs:** The graphs for the scene length statistics (generated by ***`wow.py stats`***).
- **Scenes:** The scenes created from the dataset (generated by ***`wow.py build`***).
- **IOs:** The output IO streams per scene (generated by ***`wow.py run`***).
- **environment.yml:** The conda environment initialization file (for external libraries).
- **wow.py:** The main script.

- **conf.py:** The configuration file (probabilities and cities sizes).

3 Modules Structure

The hierarchical structure of the modules used in the project (information about each module – in the next chapter):

- The ***Scene*** object is the most important object in the algorithm. It consists of the ***World*** object, the ***Avatar*** and ***Guild*** objects that participate in the scene.
- The ***World*** object consists of 3 ***Continent*** objects (Kalimdor, Eastern Kingdoms, Outland).
- The ***Continent*** object consists of all the ***Zone*** and ***Location*** objects in that continent.
- The ***Zone*** object represents an in-game zone, its boundaries and the ***City*** objects in it.
- The ***City*** object represents an in-game city/instance and its boundaries.
- The ***Location*** object represents a 60x60 meters block in the game map. It contains a collection of the ***Avatar*** objects in it.
- The ***Guild*** object represents an in-game guild. It contains a collection of the ***Avatar*** objects in it.
- The ***Avatar*** object represents an in-game avatar and consists of its places and guild ***Changes*** objects (of the scene).
- The ***Changes*** object represent the series of changes of some variable over time.



4 Modules

The modules used in the project, and their important methods.

4.1 The Location (location.py)

The location object represents a 60x60 meters block in the game map.
It has a unique ID: LO_{continent-name's first letter}_{x}_{y}.
The location knows in which x, y, zone and city (if any) it's located.
The location also maintains a set of the avatars in it at any given time.

manhattan_to(next_loc):

Returns the Manhattan distance to ***next_loc***, and two queues representing the x-axis and y-axis steps to be taken from the current location to ***next_loc***.

4.2 The Place (place.py)

Place is an abstract class, represents a rectangle of locations, which implements the function ***get_random_location(prev_location)***.

4.3 The City – subclass of Place (city.py)

The city object represents a city or an instance in the game map.
There are 4 types of cities with different sizes:

- Capital – 3x3 locations.
- Major City – 2x2 locations.
- Minor City – 1x1 locations.
- Instance – 2x2 locations.

These sizes can be modified in conf.py.
The city knows in which zone and continent it's located, its type, boundaries, and name (if named).

get_random_location(prev_location):

Returns a random location in the city (prev_location is not used).

4.4 The Zone - subclass of Place (zone.py)

The zone object represents a zone in the game map.
The zone knows in which continent it's located, its boundaries, name, a set of neighboring zones, and lists of the cities in it, by type.

get_random_location(prev_location):

Returns a random location in the zone based on prev_location:
If the previous location is a city in this zone, with probability ***P_SAME_CITY*** the returned location will be a random location in this city.
Otherwise, with probability ***P_CAPITAL***/***P_MAJOR_CITY***/***P_MINOR_CITY***/***P_INSTANCE*** choose a random location from a random city in this chosen type (tested one by one in that order).

Otherwise – return a random location in this zone.
We used the next values, and they can be modified in conf.py:

```
P_SAME_CITY=0.5
P_CAPITAL=0.2
P_MAJOR_CITY=0.15
P_MINOR_CITY=0.03
P_INSTANCE=0.3
```

4.5 Continent (continent.py)

The continent object represents a continent in the game map.
The continent knows its boundaries, name, and all the zones, cities, and locations in it.
The continent initializes its Zone and Location objects, and their connections, using ***Maps/{continent}.csv*** for each continent.

4.6 The World (world.py)

The world object represents the game world map.
The continent knows its boundaries, name, and all the zones, cities, and locations in it.
The world initializes the 3 continents, all the City objects (using ***Maps/cities.csv***), and registers the neighbors for each zones (using ***Maps/neighbors.txt***)

reset():

Clears the avatars sets in every location in the world (used for reusing the same world in a new scene).

4.7 The Changes (changes.py)

The changes object is used to capture series of changes of values over time.
We used this object to represent the avatar's guilds and places throughout the scene. This data structure allows getting the current value efficiently.
The changes object has an internal virtual-clock synchronized with the virtual-time of the scene.

register_change(vtime, val):

Captures the change in the virtual-time ***vtime*** to be the value ***val***.
The calls must be ordered in ascending virtual-time.

get_next_val():

Increments the internal virtual-clock and returns the current value.

4.8 The Guild (guild.py)

The guild object represents a guild (group) of avatars in the game.
It has a unique ID: GO_{dataset's guild-id}.
The guild maintains a set of the avatars in it at any given time.

4.9 The Avatar (avatar.py)

The avatar object represents an avatar in game.
It has a unique ID: AO_{dataset's avatar-id}, and a unique device name: A_{dataset's avatar-id}.
The avatar maintains an internal clock (seconds from the beginning of the scene), its current location and guild, and a future-path (next locations for the current virtual time).
Location=None means that the avatar is offline.
The avatar is initialized with its guild-changes and place-changes objects for the current scene.

`_update_future_path()`:

Sets the future-path to a 600 locations queue (for 1 virtual-time) according to the current location and next place (from the place-changes).
If the avatar should be offline, stay offline for the virtual time.
If the avatar just got online – set its current to a random location in the next place.
Get the last location (location in 1 virtual-time) - a random location in the next place.
If they are the same location – stay at the same place for the virtual-time.
If they are in the same zone or in neighboring zones – make a random Manhattan path between the locations. If the current location is in a city – start with staying at the current location for 3 minutes.
Otherwise – assume it's a portal and stay half a virtual-time at the current location and half a virtual-time at the last location.

`step()`:

This function advances the avatar's state by one second.
It sets the avatar's location to the first location in the future-path and increments the internal clock.
It fills the future-path if it was empty (using ***`_update_future_path()`***), and updates the avatar's guild.

`generate_io()`:

Simulates and returns the IO requests the avatar makes according to the current state.
Each call (should be once per second) the avatar requests the following objects:

- Itself
- Its location
- All the avatars in its location
- Its guild (if it's in one)
- All the avatars in its guild

Every object is requested at most one time each call.

4.10 The Scene (scene.py)

The scene object represents a running game.
It maintains a World object, an internal clock, and initializes all the scene's avatars and guilds.
The scene runs until its time_limit is reached (if there is no such limit, until the end of the scene).

The scene is built from the ***Scenes/scene{scene-number}.csv*** file, specifying the changes in avatar's places and guilds for any given virtual-time.

step():

This function advances the game's state by one second. It does it by advancing every avatar's state in the game by one second (***avatar.step()***).
It also increments the scene's internal clock.

generate_io(output_file):

Simulates the IO request made by all avatars according to the current game-state.
Writes the IOs to the ***output_file***.
This function should be called once per second.

run(keep_output):

Run the entire scene, second by second, and generate the IOs.
Each second of the scene the function calls ***step()*** then ***generate_io(output_file)***.
An output file is created for every consecutive 10 minutes (1 virtual-time).
The output files are saved as ***{Output-folder}/Scene{scene-num}/scene_{scene-num}_{first-minute}-{last-minute}.txt***.
If ***keep_output*** is false – clear ***{Output-folder}/Scene{scene-num}*** first.

The system updates location and guild objects via *WRITE* operations: location objects when an avatar moves to a new location, and guild objects when avatars join or leave a guild. Guild updates are more frequent for larger guilds, as they depend on the ratio of guild members to total avatars. Set ***include_writes*** in conf.py to activate writes.

5 Scripts

The script files contain the logic for the actions in the project, called by the main script wow.py (in the next chapter).

5.1 stats_calc.py

calc_stats(data_path, output_folder, show, min_day_records, max_gap_minutes)

Calculates statistics about the WoWAH dataset (from ***data_path***):

- Gaps – consecutive records with a time-gap longer than ***max_gap_minutes***.
- Bad days – days with less records than ***min_day_records***.
- Missing days – days without any record.
- Scene lengths – A scene is a consecutive period without any gaps (as defined above).

The script generates 3 graphs of the scene lengths (under ***output_folder***):

- Bar graph: ***bar_{max_gap_minutes}.png***
- Histogram: ***hist_{max_gap_minutes}.png***
- Cumulative Histogram: ***longer_than_{max_gap_minutes}.png***

If ***show*** is true – show the graphs to the user.

5.2 scenes_build.py

build_scenes(data_path, min_scene_minute_len, max_gap_minutes)

Builds scene files based on the WoWAH dataset (from ***data_path***).
A scene is a consecutive period longer than ***min_scene_minute_len*** without any gaps. Gaps are consecutive records with a time-gap longer than ***max_gap_minutes*** (the first gap longer than 120 minutes in each Thursday is ignored – due to the weekly maintenance).
Each scene is saved to ***Scenes/scene{scene-number}.csv***, while each row in it is of the following format: ***virtual-time, avatar-id, guild-id, place***.
Creates the file ***Scenes/scenes_summary.txt*** contains for each scene created the duration, start time and end time.

5.3 maps_build.py

create_maps(show)

Creates the continents maps divided into zones.
For each continent read ***Maps/{continent}.csv***, which contains the boundaries of all the zones in the continent.
Generate ***Maps/{continent}.png*** - visual representation of the continent, and
Maps/{continent}.pickle - numpy representation of the continent.
If ***show*** is true – also show the visual representation.

5.4 cities_build.py

build_cities(seed)

Creates the city locations on the map (without overlaps).
For each continent read ***Maps/{continent}.csv***, which contains the number of cities for each city-type, then generate random location (with ***seed***) for them in their zone.
If a city appears in the dataset – also save its name.
Save all the cities locations to ***Maps/cities.csv***.

5.5 scenes_run.py

run_scenes(scene_nums, output_folder, keep_output, compress, num_procs, seed, minutes_limit, debug_test, debug_avatar_ids)

Runs the scenes in the ***scene_nums*** list with ***num_procs*** processes, and with the random ***seed***.
The output files will be saved under ***{Output-folder}/Scene{scene-num}/***. If ***keep_output*** is false – clear this directory first.
The scenes will run for ***minutes_limit*** minutes (or until the end, if no such limit).
If the ***debug_test*** is true – save testing data in ***{output_folder}/test_data_scene{scene-num}.pickle***.

If there are avatar-ids in ***debug_avatar_ids*** that appear in the running scene – create a gif following the paths of these avatars.

If ***compress*** is specified – compress the output using gzip (compress level = ***compress***).

5.6 debug_test.py

test_scene(scene_num, input_folder)

Tests that every IO generated by the scene (in ***input_folder/Scene{scene-number}***) should have been generated. Uses the testing data (created by the scenes_run.py) from ***{output_folder}/Scene{scene-number}/test_data_scene{scene-num}.pickle***.

5.7 io_multiply.py

multiply_scenes(scene_nums, input_folder, output_folder, compress, factor, seed, num_procs, aids)

Creates a multiplied IO (saves under ***output_folder***) from the outputted IOs of the scenes in the ***scene_nums*** list with ***num_procs*** processes. It is used to increase the number of IOs/second.
Read the IOs from ***input_folder/Scene{scene-number}***.
Multiply ***factor*** times the IOs generated by the ***aids*** avatars. So, these IOs will be generated ***factor*** times each second.
If ***aids*** is empty – multiply every IO created by every avatar.

If ***compress*** is specified – compress the output using gzip (compress level = ***compress***).

6 Commands

The main script, wow.py, supports the commands: download, stats, build, maps, cities, run, test and multiply (running examples in the next chapter).

6.1 download

Download the WoWAH dataset.

usage: wow.py download [-h]

-h, --help	show help message and exit

6.2 stats

Show statistics of gaps and scene lengths in the WoWAH dataset.

Calls calc_stats() from stats_calc.py.

usage: wow.py stats [-h] [-d PATH] [-r RECORDS] [-g MINUTES] [-o PATH] [-p]

-h, --help	show help message and exit
-d PATH**, --dataset** PATH	path to dataset folder (default=/nfs_share/storage-simulations/org-traces/WoWAH)
-r RECORDS**, --records** RECORDS	minimum records in day (default=135records)
-g MINUTES**, --gap** MINUTES	maximum minutes gap allowed in-scene (default=25minutes)
-o PATH**, --output** PATH	output graphs folder (default=./Graphs/)
-w, --show	show the graphs

6.3 build

Build the scenes' csv files from the WoWAH dataset.

Calls build_scenes() from scenes_build.py.

usage: wow.py build [-h] [-d PATH] [-l MINUTES] [-g MINUTES]

-h, --help	show help message and exit
-d PATH**, --dataset** PATH	path to dataset folder (default=/nfs_share/storage-simulations/org-traces/WoWAH)
-l MINUTES**, --length** MINUTES	minimum minutes to create a scene (default=1440minutes, a day)
-g MINUTES**, --gap** MINUTES	maximum minutes gap allowed in-scene (default=25minutes)

6.4 maps

Creates the continents maps divided into zones.

Calls create_maps() from maps_build.py.

usage: wow.py maps [-h] [-p]

-h, --help	show help message and exit
-w, --show	show the continents' maps

6.5 cities

Creates random city locations (Maps/cities.csv).

Calls build_cities() from cities_build.py.

usage: wow.py cities [-h] [-s SEED]

-h, --help	show help message and exit
-s SEED**, --seed** SEED	seed for random location of cities (default=0)

6.6 run

Run and generate IOs of the scenes.

Calls run_scenes() from scenes_run.py.

usage: wow.py run [-h] [-p PROCS] [-t] [-g AVATAR [AVATAR ...]] [-s SEED] [-l MINUTES] [-o PATH]
[-c [0-9]] [-k] SCENE [SCENE ...]

SCENE [SCENE ...]	run and generate IO from these scenes
-h, --help	show help message and exit
-p PROCS**, --procs** PROCS	number of processes to use
-t, --test	collect extra information for testing
-g AVATAR **[**AVATAR**...], --gif** AVATAR **[**AVATAR**...]**	create gif follows these avatars' path
-s SEED**, --seed** SEED	seed for random steps of the avatars in the scenes (default=scene_num)
-l MINUTES**, --limit** MINUTES	time limit in minutes for scene
-o PATH**, --output** PATH	output folder path (default=./IOs/)
-c [0-9], **--compress** [0-9]	output compression level (defualt=5), no compression if not specified.
-k, --keep	don't empty the output folder before running

6.7 test

Test that the IOs generated should've been generated.

Calls test_scene() from scene_test.py.

usage: wow.py test [-h] [-i PATH] SCENE

SCENE	scene number to test
-h, --help	show help message and exit
-i PATH**, --input** PATH	input folder path (default=./IOs/)

6.8 multiply

Multiply each IO to increase the number of IOs/second.

Calls multiply_scenes() from io_multiply.py.

usage: wow.py multiply [-h] [-p PROCS] -f FACTOR [-a AVATAR [AVATAR ...]] [-s SEED] [-o PATH] [-c [0-9]]
[-i PATH] SCENE [SCENE ...]

SCENE [SCENE ...]	scene numbers to multiply
-h, --help	show help message and exit
-p PROCS**, --procs** PROCS	number of processes to use
-f FACTOR**, --factor** FACTOR	multiply each IO by this factor
-a AVATAR **[**AVATAR**...], --avatars** AVATAR **[**AVATAR**...]**	avatars to be multiplied
-s SEED**, --seed** SEED	seed for time randomization (default=0)
-o PATH**, --output** PATH	output folder path (default=the input path)
-c [0-9], **--compress** [0-9]	output compression level (defualt=5), no compression if not specified.
-i PATH**, --input** PATH	input folder path (default=./IOs/)

7 How To Run

7.1 Install the environment:

The code is written for Python 3.7+, and requires the matplotlib, numpy, tqdm, pandas and wget libraries.

You can install the libraries yourself or use the provided environment.yml file which can be helpful to create the python environment easily, with:

```
conda env create --file environment.yml
```

And then activate the environment, with:

```
conda activate wowpy
```

7.2 Download the dataset:

Make sure you have downloaded and extracted the WoWAH dataset (Download Link).

7.3 Create the scenes from the dataset:

For the first time you use the program you need to build the scenes from the dataset, so you be able to use consecutive periods of data from the dataset, run:

```
python wow.py build
```

Run the following to set default dataset directory:

```
python wow.py build --dataset my_dataset_path
```

If you want to change the minimal scene length to be 10 hours (600 minutes), and the default gap threshold to be 35 minutes, run:

```
python wow.py build --length 600 --gap 35
```

This command creates csv files in Scene directory, each file represent the data collected in one consecutive period. The command also create summary file contains information about the scenes start and end times.

For more information see here.

7.4 Add random locations for cities:

In order to add the cities' locations to the map, run:

```
python wow.py cities
```

This command creates the file cities.csv in Maps directory which contains the locations of the cities in each zone of the map.

For more information see here.

7.5 Simulate the IOs of a scene:

Now, after creating the scenes and initializing the map, we ready to simulate a scene.
For example, simulate scene 4, run:

```
python wow.py run 4
```

The default output directory is IOs, but you can change it with:

```
python wow.py run 4 --output my_output_directory
```

By default, we run the entire scene, if you want to run only the first 45 minutes of the scene run:

```
python wow.py run 4 --limit 45
```

If you want to compress the output files with gzip, run:

```
python wow.py run 4 --compress
```

You can run more than one scene, and specify the number of processes to use in order to run the scenes in parallel:

```
python wow.py run 4 10 34 52 --procs 3
```

This command creates scene directory in the output directory. In the scene directory it creates txt (or txt.gz if compression used) files, each file contains the IO requests for 10 minutes.

For more information see here.

7.6 Multiply the IOs of a scene:

Now that we have the IOs for specific scene we can multiply the request to be more frequent.

For example, multiply the requests of scene 4 by factor of 10, run:

```
python wow.py multiply 4 --factor 10
```

If you want to multiply only the requests done by the avatars 16, 3, 9 (the requests done by other avatars will be discarded), run:

```
python wow.py multiply 4 --factor 10 --avatars 16 3 9
```

The default input and output directory is IOs, but you can change it with:

```
python wow.py multiply 4 --factor 10 --input
my_input_directory --output my_output_directory
```

If you want to compress the output files with gzip, run:

```
python wow.py multiply 4 --factor 10 --compress
```

You can multiply more than one scene, and specify the number of processes to use in order to multiply the scenes in parallel:

```
python wow.py multiply 4 10 34 52 --procs 3 --factor 10
```

This command creates txt (or txt.gz if compression used) files in the scene directory, each file contains the multiplied IO requests for 10 minutes.

For more information see here.