The Emissary

or: the Munchkins Guide to Fallen London

Naoh

March 31, 2021
Version 0.1

> With hundreds of items and thousands of storylets for the creation and conversion of items, the economy of Fallen London can be thought of as a great web where finding the best pathes is difficult.
>
> Just like his namesake on *Saviour's Rocks*, the Emissary is willing to guide you to your goal, be it *Echos*, *Hinterland Scrips* or *Searing Enigmas*.

Fallen London is a great text based browser game by Failbetter Games. It is highly regarded for its world-building, but does contain repetive grinds as well.

The game world is the subject of multiple websites (beyond the game itself[1] and its excellent wiki[2] (on which much of the project is based), such as

- `https://thefifthcity.fandom.com/`
- `https://saint-arthur.tumblr.com/`

This project is not about the game world. It is about grinding items (and, to a lesser extend, some qualities) for **endgame characters**. It is meant to discover the most efficient repeatable ways to grind items with no regard for roleplaying.

1 Items (and spendable qualities)

Items are things which can be created (or acquired, but that is merely a fluff difference) and spend in Fallen London. I will only concern myself with items which can be created ad infinitum, and mostly with items which also can be spent in repeatable actions.

Item consumption and production by actions is (almost always) linear. Take the action ten times, and it will consume ten times as much and produce ten times as much.

[1] `fallenlondon.com`
[2] `fallenlondon.wiki`

1.1 Item-like qualities

Some qualities (that is to say, traits about your character) also are fundamentally linear. Tribute, for example, tracks the goodwill of a certain power in the Neath, and works totally like an item. Other qualities work almost like an item, but are capped to some value. Favours are an example here.

Some qualities are what is called pyramidical on the wiki, meaning each level increment costs an increasing amount of Change Points. Here, both the wiki and this project generally track the Change Points which are spent to increase the quality. The almost all actions are linear in Change Points, repeat an repeat an action which gives you one Change Point of *Casing...* ten times, and you will gain ten Change Points of *Casing...* The fact that a grand total of 1 CP of Casing... is level one, and 10 CP correspond to level four (or something) is mostly irrelevant for this consideration.

In the parser, the line

```
Casing... is increasing... (+4 CP)
```

is in effect translated to

```
You've gained 4 x CP: Casing...
```

Sometimes, an action resets a pyramidical quality. In these cases, I have converted the reset to a loss of Change Points based on either the minimum requirements for the action.

Some actions involve checks based on the very quality which will be reset on success. Naturally the sweet spot is dependent on the action cost to raise the quality. The easiest way to solve this would be to have a generator script which writes an action for every possible value of the quality up to $p_{success} = 1$. Happily, the actions involved (*Hunting dangerous prey*, *The big score*) seem rather non-competitive for late game grinds, so I ignore them for now.

Qualities which are technically spendable (e.g. *Dreams* in the *Nadir*) but not in repeatable, competitive actions are not implemented.

1.2 Menaces

Menaces are pyramidical qualities which have a negative effect when a character accumulates to many Change Points in them. As this tool generally treats gains as desiable and losses of items or CP as a liability, I decided just to invert them while parsing. Nightmares is increasing (+2 CP) thus becomes "You have lost 2x CP Loss: Nightmares". This way, the costs to compensate two CP of Nightmare gain will be included in the action. This is a bit of a hack but helps to keep the program logic simple.

1.3 Meta-Qualities and generator scripts

Some qualities can not be accumulated in a linear (or pyramidical) fashion. *Cover Identity: Elaboration*, for example, requires increasingly unlikely Skill Checks as it increases. To be able to handle this, I have introduced Pseudo-Items (or Pseudo-Qualities, I do not differenciate between these) to represent a specific level of them. If you need a *Cover Identity: Elaboration* at 5 for an action, and the cover Identity is reset by the action, I would write

You've lost 1x Meta: Cover Identity: Elaboration 5

A helper script, cover_identities_generator.py, can be used to automatically generate actions to transform Meta: Cover Identity: Elaboration 4 into Meta: Cover Identity: Elaboration 5 and so on based on the character attributes as given in char.py.

Somewhat different, in the Kitchen at Station VIII, you can transform many Meals (or Drinks) into each other according to specific rules, thereby changing your *Current Culinary Concoction* quality along various enum-eque values. I have solved this by introducing items Meta: Current Culinary Concoction *N* which are created and consumed by the various actions as you transform your Concoction.

Hunting and Searching in Parabola is a difficult topic that will be discussed later. The likely costs to hunt (or search) according to (somewhat) optimal strategies are automatically calculated from a python script. However, I did not want to include all the side effects in that script. Instead, I have actions which require stuff like Meta: Parabola: Hunt shark (which is gained some the auto-generated action) and convert it into the proper rewards.

The other main use for Meta-Qualities is to track mutually exclusive Guard-type requirements (see below). You can only build one station per station, for example, and different ones are required for different actions. The file char.py includes a dictionary called choices, which records the choices your character made regarding such decisions. A script choice_specific_generator.py will auto-generate an action file which will give out some meta items for free. An action which requires you to have a Feducci statue might then consume 1 x Meta: Ealing Garden Statue: Feducci. If you have it, that is a Reactant (see below) which costs zero actions to acquire, and if you do not have it, it is a Reactant you are unable to acquire, e.g. which costs an infinite amount of actions. Note that choice values of "*" in char.py will allow you to assume that you have any and all possible values of that choice.

The generator scripts included so far are:

- bundles_generator.py: a script which searches for values of Meta: Unpredictable Treasure *n–m* and Meta: Opening a Bundle of Oddities *n–m* in other actions and generates the actions converting these in rewards based on lists/*.csv[3].
- choice_specific_generator.py: Will make Choice: *something* items available based on the choices variable in char.py.
- cover_identities_generator.py: Will generate actions to increase the *Cover Identity* qualities based on the attributes in char.py.
- hunting_generator.py: Uses the simulation framework in sim/sim.py to figure out optimal strategies to hunt and search in *Parabola* given the attributes in char.py.
- poet-laureate_generator.py: Writes actions to increase your *Poet-Laureate* quality. This is so far the only case I have included non-repeatable actions, because grinding P-L takes very long and the actions needed are nontrivial to calculate (unlike, say *A Scholar of the Correspondence*).

[3]In retrospect, I should have parsed the wikitext instead of the HTML tables.

- `wiki_api/fetch_items.py`: This skript tries to fetch item infos from `fallenlondon.wiki` and parses the Bazaar and Upper River buying and selling prices to generate buying/selling actions.[4]

All generator scripts write an action file in `actions/_gen_`*`something`*`.txt`. They have to be run manually after changing `char.py` or adding actions giving new bundle ranges (etc).

1.4 Echos (or Scrip): Is there a default grind?

Echos are the in-game currency used by the Bazaar, and are mostly created by grinding sellable items. The wiki article on money making already contains an excellent guide on Echo Grinding, so I will not spend to much time on it.

There are two possible philosophical views on Echos.

One is to view them (or their quantum, the Penny) as just another item type. Instead of converting honey into romantic notions, you convert it into Echos (by selling it) and then the Echos into other items (by buying them). The only two special things about that process are that it takes no actions to buy or sell something (at the Bazaar), and that the exchange rate is usually quite unfavorable, as the prices you get for selling stuff are mostly about half what you would pay for buying.

For buying stuff at the bazaar, we take exactly that view. Each item buyable at the bazaar is converted into an action (which cost no action point) which converts the appropriate amount of Pennies into the item and is labeled `Buy at the bazaar:` *`itemname`*. Additionally, we add a configurable action `Grind for` *`value`* `EPA`. Sometimes, the best way to get an item is indeed by grinding echos and paying for it.

The other view is to see Echo grinding as the default occupation. This recognizes that Echos as the only in-game pathways to lots of items, including Goats and Cider. In the Ambitions storylines, you will also sometimes get stuck for weeks[5] while grinding the 10k Echos required to buy items.

For selling stuff, things are more complex. This project is not about optimizing echo grinding, but item grinding. However, there is a reason to include the prices you get for selling. Often, our grinds have side effects, both bad (Menaces, item losses) and good (additional item gains). Naturally, we include the costs for the bad side effects. For the good side effects (mainly item gains), we have two options. We could either say *Well, while grinding for item X, you did accumulate lots of Y, but we just ignore that* or we could say *You are probably grinding for Echos all the time anyhow, so if we sell the Y at the Bazaar, this will save you some actions, reducing the effective item cost.*

In the current version of the Emissary, you can specify your default action sink using the `--background` option. As an argument, give the item name (e.g. Echo or Hinterland Scrip). This will check for the best grind for that item, and rebate all other grinds by the amount they further that grind.

[4] Note that when you run this script, or otherwise update based on wiki content, you should also run `wiki_api/wiki_contributors.py` to make sure that the list of contributors is up to date (as required by CC-BY-SA).

[5] Or even *months* if you like to spend longer than the 3 hours and 20 minutes (6:40 for exceptional friends) away from the fifth city for less important stuff like sleep, work, school, social contacts.

2 Actions (or Storylets)

The first fundamental resource in Fallen London are action points. One action point is generated every ten minutes up to a limit of 20 (or 40 for paying users). They are spend for most actions, from fluff only over item creation and item conversion.

Actions can be categorized by avalability. Some are always available (subject to certain static requirements, see below). Some are seasonal (and will not be covered by this project), some are limited by real time (e.g. one a week, and will mostly also not be covered here) and some are limited by the other fundamental resource: **Opportunity cards**. One card is created every ten minutes up to your deck size of six (or ten). Some locations have special sets of cards, some of which allow infinite draw.[6]

Favours are item-like qualities which can mostly be grinded using cards. With the notable exception of the card *A visit*, favour-yielding cards take one action and may consume or produce about half an Echo worth of resources.

Rather than listing all Favour-grinding cards, I have just added faction specific Meta-cards which will allow to gain a favour for an action. **By setting the `--favours` option, you can enable these.** Otherwise, favours will only be available through generally expensive static storylets such as Pinewood shark hunting, or even not at all.

For the rest of the cards, the `-c` flag allows you to view how much the cards implemented here would help you with your grind. Also, you can make all cards available freely with the `-C` flag.

2.1 Prerequisites: Guards and Reactants

Some actions have prerequisites, e.g. certain qualities or items required to do the action. Some of these are what I would like to call **Guards**: things that are not consumed by the action. For example, you need to be a person of some importance to do the Eavesdropping action in Spite. This project concerns itself little with these. If an action requires you to have A Scholar of the Correspondance 21, and you don't have that, there is nothing stopping you from spending a finite amount of actions gaining that, and afterwards, you can freely use that action. But sometimes, requirements are mutually exclusive. A single character can only finish one Ambition (and gain only some of the possible rewards), have one Profession (through that is changable), one statue in Ealing Gardens and so on. Such actions will consume a special Pseudo-Item named `Choice: Choicename`. The source for these Pseudo-Items is a file written by `generators/choice_specific_generator.py`. Depending on your configuration of `char.py`, the Pseudo-Item will either be available (without any action cost) or not at all.

For the others, I would like to borrow the term **Reactants** from chemistry, meaning that which is consumed by a reaction. In almost all cases, the action which will consume something also has that thing as a prerequisite. Unlike Guards, Reactants are meant to be precisely tracked by this project. They are parsed from indented text file lines like TODO.

[6] In the actions folder, actions which require a card are marked with `Card: Cardname`. Exception: the cards drawn in the camp of the Clay Highwayman are not marked thus, as there is only a very limited amount of drawable cards, and the only non-discardable ones are also highly profitable.

2.2 Products

The motivation to play an action is to read the text describing the outcome. The reason to play an action for twenty times is to read the text describing the rare success. The reason to play an action a thousand times is because of its material rewards. For this guide, this means items or item-like qualities (see above).

The `-g` flag specifies which product you want to grind. Alternatively `-G` will find the best grind for any known items.

3 Algorithm considerations

3.1 The original approach

Note: This section describes the first, greedy approach. It is no longer used.

Fundamentally, the main script tracks the minimal cost (in actions) to create each item from the scratch. Take an action A which generates N_X times item X. Take its action cost C_A (mostly, but not always, 1) plus the cost to create all Reactants (for minimal costs). Divide that by the N_X and you have the effective action cost $T_{A,X}$ you spend per item X via when creating it via action A.

Minimize $T_{A,X}$ over all actions A which generate any item X, and you have the minimal cost M_X.

Problem 1: Strictly speaking, you would have to know the minimal costs of each item to calculate the minimal costs of each item.

The script tries to solve this iteratively. We initialize the M_X to infinity for all X. Then, we loop over all actions, updating on any M_X as we go along. First, we will only consider actions which to not require Reactants, as any reactants cost is still infinite. By and by, we will assign finite costs to more and more reactants, and eventually loop though all actions without having to update M_X for any X. Then we are done.

Problem 2: That looks like it might converge slowly

Assume we have a two item cross category trade on the most efficient pathway: One action converts a Brilliant Soul into two Tales of Terror!!, and another converts one Tale into two Brilliant Souls. Obviously, each item will cost one action to create (if we have some spare Reactants already).

Let's assume that initially, we only can create a Brilliant Soul for 65 actions. The algorithm will first update the cost of one Tale to 33 actions. Then it will update the cost of the Brilliant Soul to 17 actions. In general, each update will go from $1 + \epsilon$ to $1 + \frac{\epsilon}{2}$.

A bit per iteration might be acceptable (there are only so many bits in the mantissa of a floating point number), but now consider the case where we convert N items into N+1 items. Suddenly, the update goes from $1+\epsilon$ to $1+\frac{n}{n+1}\epsilon \approx 1+\left(1-\frac{1}{n}\right)\epsilon$. Now consider that the actions are processed one by one in some (essentially) arbitrary order, so we might have to process all the other action before we can update the one rule which depends on our previous update.

Please note that in practice, this did not seem to occur. While there are cycles (such as the cross category conversions), not all of the actions of a cycle are the most efficienct actions to grind their product.

Problem 3: Some actions produce multiple items

Do you remember the Echo section above? I mentioned that the two philosophies would become relevant. One philosophy says that you are just grinding for your item and any additional items you accumulate should be ignored.

The other philosophy assumes that the item grinding does not happen in a vacuum, but against a background of continous Echo grinding. This means by selling the byproducts, we can save some actions we would have otherwise spent grinding echos.

Problem 4: What about synergies when grinding for Reactants?

This is the fundamental limit of the greedy approach. If there was an action giving you the correct ratio of *Justificande Coins* as well as *Bessemer Steel Ingots*, it might not be the most efficient action to grind either of these products, but could still be the most efficient action when grinding for *Railway Steel*, which will consume both of these.

This was not solvable within the greedy algorithm and thus called for a better approach.

3.2 An approach based on Computer Science

After careful consideration[7], I belive that the chosen *abstraction*[8] of Fallen London can be formulated as a well known CS problem.

Consider a number n of possible actions (or storylets) and a number m of possible items. First, assign each action an integer number between 0 and $n-1$ and each item a number between 0 and $m-1$.

Let $A_{i,j}$ be the changes the action j will apply to item i. For example, if action 5 consumes 100 elements of item 8, $\mathbf{A}_{8,5} = -100$. $\mathbf{A}$ will be a matrix over $\mathbb{R}^{m \times n}$.

Note that each column of $\mathbf{A}$ represents the item changes affected by running the action corresponding to the column index once.

Let c_j be the action cost of action j. Overall, we have $\mathbf{c} \in \mathbb{R}^n$.

The user has a shopping list. They want to grind b_i items of type i. Overall, we have $\mathbf{b} \in \mathbb{R}^m$.

Let x_j represent the number of times we run action j. $\mathbf{x}$ will be a vector of the vector space $\mathbb{R}^n$. Almost[9].

Putting it all together, we could state our problem. Minimize the total action cost:

$$mathbfc^T\mathbf{x}$$

under the constraint that you grind all the items you need

$$\mathbf{A}\mathbf{x} \geq \mathbf{b}$$

[7]That is to say, after I considered my (greedy) implementation to be complete and was just writing the manual, some ten days ago.

[8]A term used in-game for the removal of the soul. If my model is what is left after the removal of the soul of Fallen London, or rather its very soul is probably a matter of opinion.

[9]Strictly speaking, *Fallen London* requires x_j to be a non-negative integer. However, this will complicate the problem, possibly making it NP-hard. We will retreat to the position that we want to find the asymptotically best grind, and that real numbers are fine for that. We will respect the requirement of x_j being non-negative, though.

and (obviously)

$$\mathbf{x} \geq 0$$

Happily, this is a well known problem in **linear programming** (or linear optimization), the so called **covering problem**.

SciPy[10] includes a function called `scipy.optimize.linprog` which will solve these problems for us. In the beginning, I used SciPy 1.1, and had some problems with algorithms converging rather slowly or not at all. This is possibly related to my lack of experience in using numerical algorithms, e.g. not preconditioning the matrix or stuff like that.

Browsing the documentation for SciPy 1.6.1, I noticed that there are newer linprog algorithms included in that release. On a whim, I spend half an hour compiling that release, which turned out to be a good investment. SciPy 1.6 can use the HiGHS library[11] to solve linprog problems, which turned out to work very well. This is why you will probably have manually install a current version of SciPy by hand (unless it should be included in your linux distribution).

3.2.1 Background grinds

The first approach to accomodate background grinds was just to include a largish number of these items (e.g. Echos) in **b**. This worked reasonably well but complicated parsing the results, which were a mixture of the most efficient grind for the background item and the grind for the item under consideration.

A better, if slightly hacky way to handle that was to remove the background item from the constraints matrix and add it to the cost function instead. This basically tells the algorithm *you don't have to keep track of (e.g.) Echos, but the total action cost will be modified by* $\frac{\Delta Echo}{EPA}$.

Care must be taken when setting the action cost worth of the background item: If it is to low, the algorithm might decide just to spend Echos and buy items at the Bazaar. Even worse, if it is to higher than the action cost of the best grind, the algorithm will be able to save actions, turning the action cost negative.

For now, the -b '*background item*' option will simply calculate the best grind for that item and use that action cost.

4 Actions file format

The actions file format is based on the gain/loss or increase/drop output lines of Fallen London (which are also used in the wiki).

Non-indented lines are action names. They are followed by one or more indented lines (starting with at least one space character) of the following types:

- Card-requirement lines

  ```
  Card: Some Card Name
  ```

- Item gain/loss lines:

[10] `www.scipy.org`
[11] `https://www.maths.ed.ac.uk/hall/HiGHS/`

```
You've lost 1 x Strong-Backed Labour
You've gained 28-34 x Whispered Hint
```

- Quality increases/decreases:

```
Making Waves is increasing... (+1-15 CP)
Connected: Benthic is dropping... (-5 CP)
```

Note that the ranges will get converted into their expected value.

Checks are not supported. If a late game character will always pass the check, the failure result is ignored, as are raw results (unless a probability for them is stated on the wiki).

For failable checks, the expected outcome (based roughly on the stats of my primary character) are provided. This means that sometimes the gains will be floating point numbers. Alternate results are considered just as likely.

5 Further resources

The text files in the top level directory contain information about limitations of the software, copyright notes, dependencies of the scripts and possible future developments. Check `sample_outputs` for some examples of how to use emissary.

6 Thanks

First and formost, thanks to **Failbetter Games** for Fallen London. And also for being ok with me putting the `actions/` files on github.

This project would not be possible without without the people on **`fallenlondon.wiki`** (formerly `fallenlondon.fandom.com`) documenting every nook and cranny of Fallen London. Virtually all of the `actions/` content is based on their efforts, either verbatim or modified so that my parser can understand it.

While I do not agree with everything on **Python**, it is a nice language to use. Its main advantage are the many available libraries – such as **NumPy** and **SciPy** – which do the heavy lifting. Also, the **HiGHS** was really helpful in providing fast solutions to my problem while allowing me to remain ignorant of the finer points of linear programming algorithms.