Team members: Nick Pampe and Nick Smith

Our final project uses data from a video game called Factorio. In Factorio, there is a very complex technology tree of craftable items that the player must balance in order to keep a factory working at maximum capacity. One of the most important aspects of this is to make sure that all products are being produced at the correct rates. Too few of a product in the middle of your supply chain will massively slow down the rate at which more complex products are created.

To get our data, we found a script that crawls through Factorio's game data and finds the crafting recipes for each item. This data is saved as a .csv and copied into the table "factorio_recipe". From here, a python gui was written that allows the user to ask for the recipe for a particular product. The result returned is not just the parts needed for the requested part, but the parts needed to make those ingredients, and the parts needed to make those sub-ingredients, and so on until you reach the most basic recipes. This would allow Factorio players to optimize their builds, as they can find the resource production needed to sustain a certain goal (i.e. create one satellite per minute).

[Data collection]

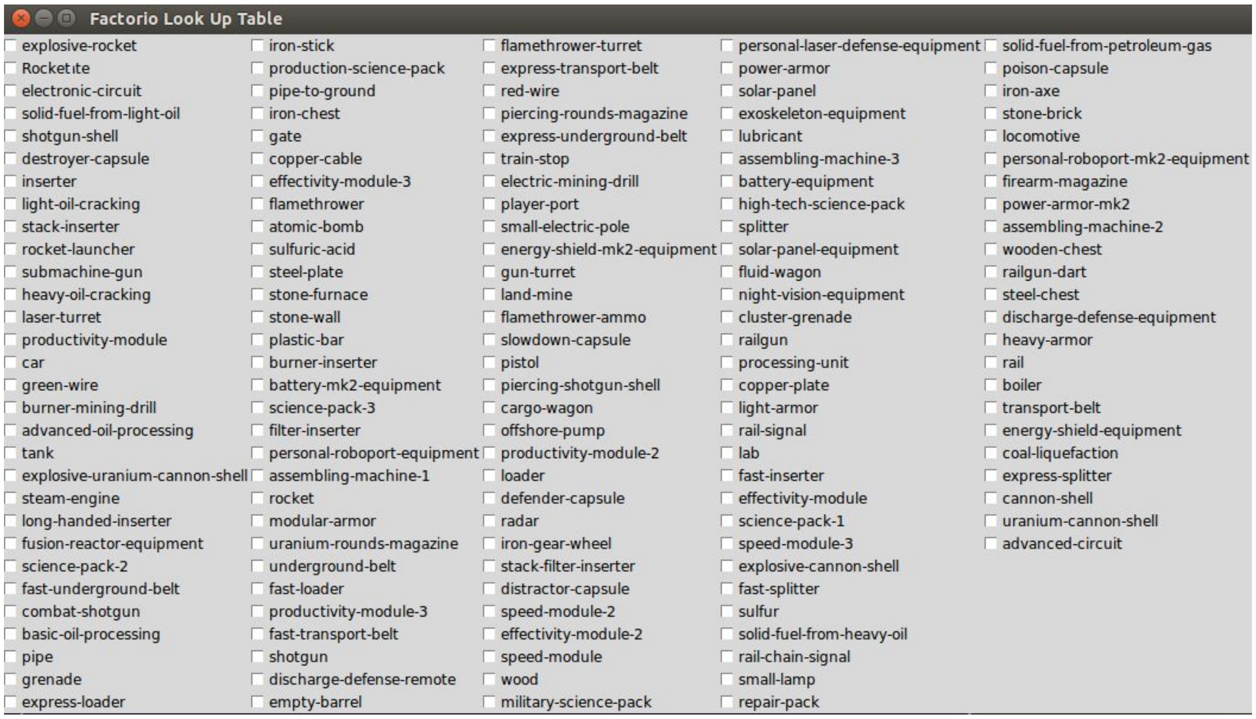
The data is acquired by a lua script based off of a script written by git user pfmoore. It iterates through every file of recipes, then every recipe in the file, and finally through every ingredient in the recipe. For each step in this iteration, the name of the recipe, name of ingredient, and number of ingredient pieces required (scaled by the number of products produced) is output to the console and a .csv file. The game files are organized in a way that makes this type of iteration very easy. Each file is a .lua with an almost XML-like pattern. Each recipe has flags for recipe name and ingredients, and some recipes have a flag for the amount of products created by a single "recipe unit" (for lack of a better term).

Nothing particularly special happens inside the loop through each game file, as the game files are only really separate to organize different types of items. Inside the loop for each recipe, the name of the product and amount of products created by a single recipe is found. The result is assumed to be 1, but if the result_count flag exists, it is read to get the proper number. Inside the loop through individual ingredients, the name of the ingredient and number of ingredients needed is read. The number of ingredients is then divided by the number of products created, to get an accurate ingredients:products ratio. Additionally, some ingredients are fluids, and have fractional amounts in most ingredients. The game data stores these amounts as ints, so to make fractional amounts work, the fluid amounts need to be divided by 10. This information is then printed to console and a .csv in the format "[recipe name], [ingredient name], [scaled ingredient amount]".

Due to the security on the flowers server, copying data from a .csv into the database requires jumping through some hoops. First of all, an empty table needs to be created with fields for recipe name, ingredient name, and amount needed. At this point, the psql \copy command is used to invoke COPY FROM STDIN or COPY TO STDOUT, which then fetches/stores the data line-by-line from a file accessible to the psql client to be imported into the table. The default delimiter for \copy is the tab character, so using a comma as the delimiter needs to be specified. Other than that, \copy is pretty self-explanatory.

[GUI designed]

The GUI (Graphical User Interface) runs from a python script that makes use of a package called tkinter**.** Useful tools, such as window frames, scroll bars, and interactive frames let the user directly influence the database without needing tedious knowledge of psql commands. In addition, the GUI can make the program much cleaner and simpler to use. Our main GUI was setup to use checkboxes to determine which items were being requested for a crafting tree look up. Interestingly, the GUI is built from a simple query to receive all the recipes. Thus, if new versions of the game are released with additional content, or mods used to add items, the GUI will update itself. An example of our main GUI is shown in the following figure.



☐ explosive-rocket	☐ iron-stick	☐ flamethrower-turret	☐ personal-laser-defense-equipment	☐ solid-fuel-from-petroleum-gas
☐ Rocketite	☐ production-science-pack	☐ express-transport-belt	☐ power-armor	☐ poison-capsule
☐ electronic-circuit	☐ pipe-to-ground	☐ red-wire	☐ solar-panel	☐ iron-axe
☐ solid-fuel-from-light-oil	☐ iron-chest	☐ piercing-rounds-magazine	☐ exoskeleton-equipment	☐ stone-brick
☐ shotgun-shell	☐ gate	☐ express-underground-belt	☐ lubricant	☐ locomotive
☐ destroyer-capsule	☐ copper-cable	☐ train-stop	☐ assembling-machine-3	☐ personal-roboport-mk2-equipment
☐ inserter	☐ effectivity-module-3	☐ electric-mining-drill	☐ battery-equipment	☐ firearm-magazine
☐ light-oil-cracking	☐ flamethrower	☐ player-port	☐ high-tech-science-pack	☐ power-armor-mk2
☐ stack-inserter	☐ atomic-bomb	☐ small-electric-pole	☐ splitter	☐ assembling-machine-2
☐ rocket-launcher	☐ sulfuric-acid	☐ energy-shield-mk2-equipment	☐ solar-panel-equipment	☐ wooden-chest
☐ submachine-gun	☐ steel-plate	☐ gun-turret	☐ fluid-wagon	☐ railgun-dart
☐ heavy-oil-cracking	☐ stone-furnace	☐ land-mine	☐ night-vision-equipment	☐ steel-chest
☐ laser-turret	☐ stone-wall	☐ flamethrower-ammo	☐ cluster-grenade	☐ discharge-defense-equipment
☐ productivity-module	☐ plastic-bar	☐ slowdown-capsule	☐ railgun	☐ heavy-armor
☐ car	☐ burner-inserter	☐ pistol	☐ processing-unit	☐ rail
☐ green-wire	☐ battery-mk2-equipment	☐ piercing-shotgun-shell	☐ copper-plate	☐ boiler
☐ burner-mining-drill	☐ science-pack-3	☐ cargo-wagon	☐ light-armor	☐ transport-belt
☐ advanced-oil-processing	☐ filter-inserter	☐ offshore-pump	☐ rail-signal	☐ energy-shield-equipment
☐ tank	☐ personal-roboport-equipment	☐ productivity-module-2	☐ lab	☐ coal-liquefaction
☐ explosive-uranium-cannon-shell	☐ assembling-machine-1	☐ loader	☐ fast-inserter	☐ express-splitter
☐ steam-engine	☐ rocket	☐ defender-capsule	☐ effectivity-module	☐ cannon-shell
☐ long-handed-inserter	☐ modular-armor	☐ radar	☐ science-pack-1	☐ uranium-cannon-shell
☐ fusion-reactor-equipment	☐ uranium-rounds-magazine	☐ iron-gear-wheel	☐ speed-module-3	☐ advanced-circuit
☐ science-pack-2	☐ underground-belt	☐ stack-filter-inserter	☐ explosive-cannon-shell	
☐ fast-underground-belt	☐ fast-loader	☐ distractor-capsule	☐ fast-splitter	
☐ combat-shotgun	☐ productivity-module-3	☐ speed-module-2	☐ sulfur	
☐ basic-oil-processing	☐ fast-transport-belt	☐ effectivity-module-2	☐ solid-fuel-from-heavy-oil	
☐ pipe	☐ shotgun	☐ speed-module	☐ rail-chain-signal	
☐ grenade	☐ discharge-defense-remote	☐ wood	☐ small-lamp	
☐ express-loader	☐ empty-barrel	☐ military-science-pack	☐ repair-pack	

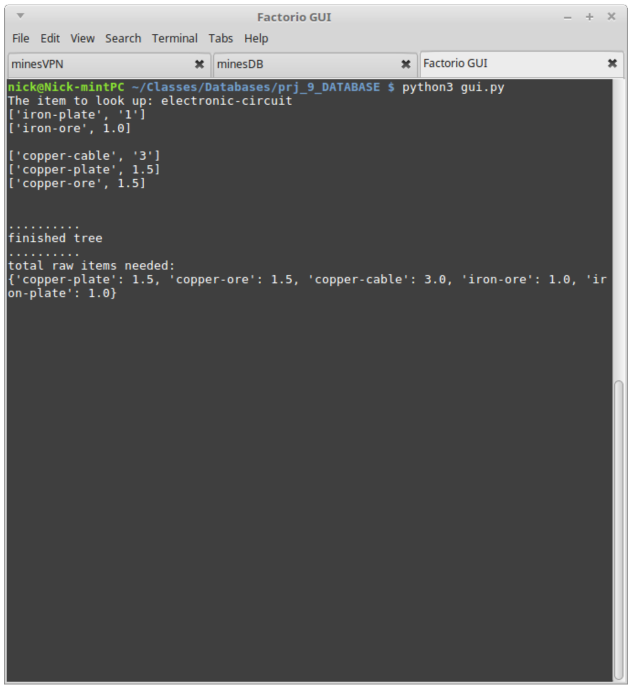
Our output is a bit messy and currently writes to the terminal. The future goals with this project are to create a visual tree, hopefully making use of python's packages such as graphviz. These packages are a simple way to visualize a tree structure, such as the building requirements used in Factorio. At this stage, we simply wanted to print the tree out to the terminal and count the total resources needed to build the requested item. This allowed us to focus more of the recursive queries and correcting the dataset. An output could look something like the following figure. In this example, we can see a green-wire was requested. Following the tree pattern, a green-wire needs one electronic-circuit and one copper-cable. Breaking down farther, an electronic-circuit needs one iron-plate and three copper-cables. The tree follows this pattern until a raw resource has been found, such as iron-ore or copper-ore. Finally, the total items are counted and displayed so that the user can quickly see was shared, bulk resources are needed. In this example, the user could produce extras copper-plates so that the system can share copper-plates between the production of the electronic-circuit and the green-wire.

[Recursive query]

With a large amount of items in the Factorio game and some recipes requiring a large amount of products, we built a recursive search algorithm. The system starts from a list of items required for the requested item. For each item, if it's not a raw recourse, the database is searched for a list of items required. The recursive algorithm is called again for each new item untill they are all raw resource. This leads the first items being broken down first. An intresting note, since each step requires a new query, the user quickly sees the output being printed though they also notice the live latency from the recursive queries.

[Examples]

One of the core parts in Factorio is the electronic circuit. This is among the first "Tier 2" product players need to make, meaning it is not made from basic parts, but rather other pre-crafted products. It requires 1 iron plate and 3 copper wires. 1 iron plate is made by smelting 1 iron ore, but copper wire is more complex. 2 copper wires are made from 1 copper plate, which is in turn smelted from copper ore. Therefore, it can be calculated by hand that one electronic circuit requires 1 iron ore and 1.5 copper ore. Querying through our program confirms this:

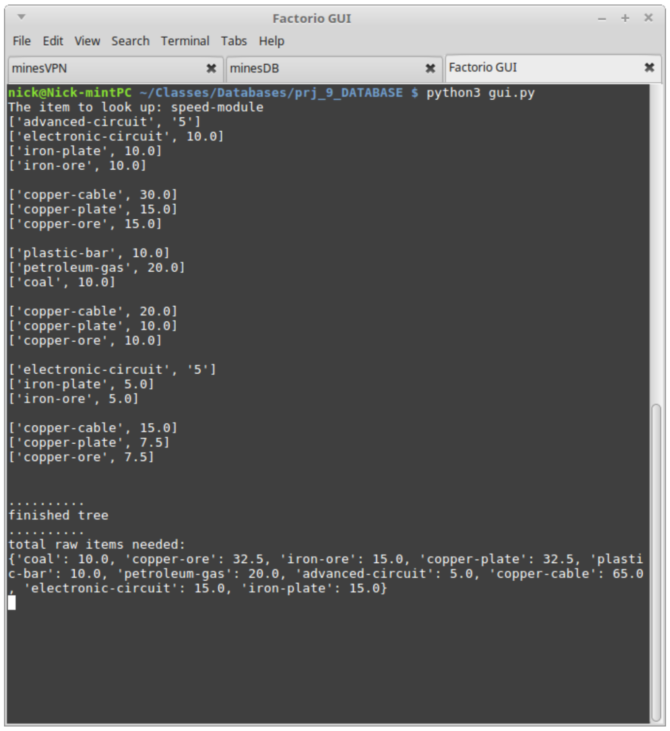


```
nick@Nick-mintPC ~/Classes/Databases/prj_9_DATABASE $ python3 gui.py
The item to look up: electronic-circuit
['iron-plate', '1']
['iron-ore', 1.0]

['copper-cable', '3']
['copper-plate', 1.5]
['copper-ore', 1.5]


..........
finished tree
..........
total raw items needed:
{'copper-plate': 1.5, 'copper-ore': 1.5, 'copper-cable': 3.0, 'iron-ore': 1.0, 'ir
on-plate': 1.0}
```

The program remains accurate for more complex recipes. The speed module is a late-game item that allows your factory to operate more quickly, at the cost of higher energy consumption and a complex build path. Because there's so many intermediate products, I'll just skip to the end and say that this requires 15 iron plates, 32.5 copper plates, and 10 plastic bars. Our program confirms this:



```
nick@Nick-mintPC ~/Classes/Databases/prj_9_DATABASE $ python3 gui.py
The item to look up: speed-module
['advanced-circuit', '5']
['electronic-circuit', 10.0]
['iron-plate', 10.0]
['iron-ore', 10.0]

['copper-cable', 30.0]
['copper-plate', 15.0]
['copper-ore', 15.0]

['plastic-bar', 10.0]
['petroleum-gas', 20.0]
['coal', 10.0]

['copper-cable', 20.0]
['copper-plate', 10.0]
['copper-ore', 10.0]

['electronic-circuit', '5']
['iron-plate', 5.0]
['iron-ore', 5.0]

['copper-cable', 15.0]
['copper-plate', 7.5]
['copper-ore', 7.5]


..........
finished tree
..........
total raw items needed:
{'coal': 10.0, 'copper-ore': 32.5, 'iron-ore': 15.0, 'copper-plate': 32.5, 'plasti
c-bar': 10.0, 'petroleum-gas': 20.0, 'advanced-circuit': 5.0, 'copper-cable': 65.0
, 'electronic-circuit': 15.0, 'iron-plate': 15.0}
```

Some of the highest-complexity recipes do not work correctly, for reasons that have not yet been identified. They correctly recurse through the sub-ingredients, but the total cost reported at the end does not give the expected value.

[Challenges faced]

One of the biggest challenges was the data-parsing script. It was written for an old version of factorio, and did not take into consideration recipes with more than one product created, or the fact that fluids are measured in smaller “units” by the game data. Having never touched lua before, I needed to learn a decent bit just to even grasp what the code was doing. Once I had that down, I was able to read through the game data (which was also written in lua) and find out how to detect when those factors affected the recipe.