Chair for System Simulation

Winter Term 2021/2022

Prof. Dr. Harald Köstler, Jan Hönig, Marco Heisig

Advanced Programming Techniques
Sheet 1 — The Factorio Build Order Optimizer

This exercise sheet is not mandatory. However, the points you get for solving the exercises on this sheet will award you bonus points for the exam.

The goal of this exercise sheet is to compute valid build orders for the video game Factorio. A build order is a list of valid events, such as "build a factory", "instruct a factory to produce iron plates", "win the game". In order to win the game, you need to produce a certain amount of some specified items. Crafting these items requires other items, particular factories or some unlocked technologies. The details of what constitues a valid build order are described in the next subsections.

No prior knowledge of Factorio is necessary, this exercise sheet should be self contained.

1 Game Entities

This section describes the game entities that your build order generator will have to deal with.

1.1 Item

An item is a game entity that can be consumed in order to craft other items, to build factories, to unlock technologies, and to win the game. For the purpose of this assignment sheet, an item is exahaustively defined by its name.
All items are described in the file `item.json`.

1.2 Recipe

A recipe has a *name* (represented as a string), a *category* (represented as another string), *energy* which is the amount of ticks necessary to execute the recipe (an integer), and a description of its *ingredients* and *products*. Both ingredients and products are a list of pairs of items and the respective amount. A recipe also has a boolean value that indicates whether it is initially *enabled*, or whether it has to be unlocked via some research first.
All recipes are described in the file `recipe.json`.

1.3 Factory

A factory has a *name* (represented as a string), a *crafting speed* (represented as a double), and a list of crafting categories. The list of crafting categories describes which recipes can be executed by the factory. The crafting speed is used to compute how many game ticks it takes to execute the recipe. The crafting time (in game ticks) is the ceiling of the product of the recipe's necessary amount of energy divided by the factory's crafting speed.
All factory types are described in the file `factory.json`.

1.4 Technology

A technology has a *name* (represented as a string), some *prerequisites* (a list of other technologies), and some *ingredients*. The ingredients are of the same format as those of a recipe. Some technologies have effects,

which unlock certain recipes. A technology can be unlocked as soon as the necessary ingredients are available. These ingredients are then consumed in the process.
All technologies are described in the file `technology.json`

2 Events

The actual simulation of the game runs by executing a series of discrete events. Each event modifies the game state in some way. An event can also be invalid, e.g., because some prerequisites or ingredients are missing. If an invalid event is encountered, the simulation aborts.

The rest of this section describes the events that may appear in a build order, and how they can be encoded as a JSON file. The actual behavior of these events in the game is described later in section 3.

A `research-event` has the following attributes:

- **timestamp** — An integer describing the game tick in which this event will be executed.
- **technology** — A string that is the name of the technology to unlock.

A `build-factory-event` has the following attributes:

- **timestamp** — An integer describing the game tick in which this event will be executed.
- **factory-id** — An integer that can be used in later events to reference the factory. The factory ID must not clash with that of any existing factory.
- **factory-type** — A string, describing the type of factory that shall be built.
- **factory-name** — A string that will be used as a description of that factory. A factory's name is only used for logging and error reporting.

A `destroy-factory-event` has the following attributes:

- **timestamp** — An integer describing the game tick in which this event will be executed.
- **factory-id** — An integer that is the ID of a previous build factory event.

A `start factory event` has the following attributes:

- **timestamp** — An integer describing the game tick in which this event will be executed.
- **factory-id** — An integer that is the ID of a previous build factory event.
- **recipe** — A string that is the name of the recipe that shall be executed by this factory for the forseeable future.

A `stop-factory-event` has the following attributes:

- **timestamp** — An integer describing the game tick in which this event will be executed.
- **factory-id** — An integer that is the ID of a previous build factory event.

A `victory-event` has the following attributes:

- **timestamp** — An integer describing the game tick in which this event will be executed.

The following example should be produced by your program for the first coal challenge:

```
[
    {
        "type":"start-factory-event",
        "timestamp":0,
        "factory-id":0,
        "recipe":"coal"
    },
    {
        "type":"stop-factory-event",
        "timestamp":60,
        "factory-id":0
    },
    {
        "type":"victory-event",
        "timestamp":60
    }
]
```

3 Game Update Rules

This section describes how the events of a build order are processed in order to advance the simulation. The game advances by in discrete steps called *game ticks*. [1] The timestamp of each event in the build order determines the game tick in which the event is executed.

3.1 Initialization

The game state contains a set of initial items, initial factories, and some goal items. These initial settings are provided by a JSON file. An example initial configuration is specified in the file `example-challenge.json`. The initial game state has a timestamp of -1, zero unlocked technologies, and exactly those unlocked recipes that are marked as enabled in the `recipe.json` file. The game state also has a list of starved factories, which is initially empty.

3.2 Game Tick

The simulator advances from game tick to game tick, until the simulation either encounters a successful victory event or fails. The following steps are performed for each game tick:

1. The game state's timestamp is incremented. If the timestamp reaches or exceeds 2^{40}, the simulation aborts.
2. All events whose timestamp matches that of the game state are grouped by their type. The group of research events is then sorted alphabetically by the name of the technology. All other groups of events are sorted in increasing order of their factory ID. All later steps execute these events in the sorted order (The sorting makes the simulator deterministic).
3. All recipes that finish executing their recipe in this game tick add their respective products to the game state.
4. All research events are executed, i.e., each technology's prerequisites are checked and its ingredients are removed from the item pool. Then the technology is unlocked, which usually means that new recipes are unlocked, too.

[1]The game Factorio executes 60 game ticks per second, but this is irrelevant for the simulator in this exercise sheet.

5. All stop factory events are executed. If a factory is stopped while working on a recipe, the work stops and the recipe's ingredients are added back to the item pool.
6. All destroy factory events are executed. Destroying a factory consists of three steps. Firstly, the factory is stopped. Secondly, the factory is removed from the game state, meaning its factory ID will be available again. Thirdly, an item of the same name as the factory type is added to the item pool.
7. If a victory event is present, the simulator checks whether the game state contains all goal items in the correct quantities. If so, the build order is correct and the simulator stops.
8. All build factory events are executed. This means that an item of the name of the factory type is removed from the item pool, and that the specified ID can now be used to reference that factory.
9. All start factory events are processed, meaning the factory will from now on execute the specified recipe. If the factory was already working on a recipe, the work is aborted and the old recipe's ingredients are added back to the item pool.
10. All factories that have finished executing their recipe in this game tick, all factories that have just been part of a start factory event, and all starved factories are sorted by their factory ID. Then, starting from the factory with the lowest ID, each such factory checks whether the item pool contains enough items to execute the factory's recipe again. If so, the ingredients are removed from the item pool and the factory continues working on its recipe. All factories for which the item pool didn't contain enough ingredients to continue executing are now the new value of the game state's list of starved factories.

If any of the checks in the simulator fails, e.g., because a build factory event was issued for which there was no item of the same name as the factory type, the simulation aborts.

4 Simplifications

Factorio veterans will have realized that the game rules in this assignment sheet are quite a bit simpler than those of the actual game. We have introduced these changes to keep the description of the exercise sheet concise, and because this sheet should be primarily about C++ and not about the intricacies of a particular video game. But for those that are curious, here is a list of the differences between our game rules and the actual game (if you don't care about Factorio, you can skip this section):

1. The logistics aspect of the game has been removed entirely. There is one global item pool. All factories can access this global item pool with zero delay.
2. The fluid mechanics have been removed. Fluids are treated just like regular items.
3. There is no electricity grid and buildings require no power. In the case of buildings that burn coal to operate, the coal requirement has been encoded directly in the recipe.
4. All buildings, items, and technologies related to uranium processing have been removed.
5. Research is instantaneous and requires no laboratories.
6. The player is modeled as a factory that cannot be deconstructed.
7. There are no biters and no pollution.

5 Factorio Server

- Create or find your group on the Factorio Server: `www10.cs.fau.de/factorio/`
- Submit your code on the Factorio Server in a zip-file, e.g. `project.zip`
- After unpacking your code (`unzip project.zip`) there should be two bash script files: `build.sh` and `generator.sh`
- Calling `./build.sh` should build your program
- Calling `./generator.sh -t target.json` should run your program, with the given target and print a detailed log to stdout as described in section 2. `target.json` will be relative path to specific JSON-file. The output from `stdout` must contain only a valid JSON log.

6 Disclaimer

This is the first year in which we will optimize build orders for Factorio instead of Starcraft. This also means that we had to rewrite all assignment sheets and our tool chain. So please forgive us in case you stumble across a bug or typo, and please also report that problem so we can fix it.

Exercise 1 — Software Architecture

Think about how you want to model the game entities, and which functions and classes you will need for your assignment. For example, you might want to have classes for enities like `item`, `factory`, or `game-state`. Then set up your project accordingly and also set up a convenient build system.

Exercise 2 — JSON Input and Output

Once you have defined your game entities, it is time to load their specification from the JSON files that we have supplied. Use an existing JSON library for C++ and make sure that you can read the files `example-challenge.json`, `factory.json`, `item.json`, `recipe.json` and `technology.json`.

Note: Do not manually convert the content of the JSON files to code. We might have to change the contents of these files later in case there are bugs or balancing problems, and your implementation should be able to adapt to such changes.

Exercise 3 — The Game Simulation

Now translate the specification from section 3 into code and check whether its behavior matches our reference implementation. After this assignment, you should have a simulator that can be invoked with a settings file and a build order, and that will determine whether that build order is valid.

Exercise 4 — Computing Build Orders

Now that you have loaded all game entities and encoded all rules, it is time to compute build orders and to score some points. We have prepared multiple challenges with increasing difficulty. The first two challenges are a warm-up exercise where we also provide you the corresponding solution. Computing a valid build order for any of the further challenges will award you 2 Points per challenge.

a) Coal (0 points) In this first challenge, all you have to do is to instruct your player to dig for coal.

b) Iron (0 points) In this challenge, you need to set up a furnace to produce iron plates. Your furnace will require both coal and iron.

c) Science (2 points) In this challenge, you need to produce your first science packs. Science packs are needed in later challenges for researching technologies.

d) Logistics (2 points) In this challenge, you have to research your first technology. This technology will then unlock the item that you need to craft.

e) Processors (2 points) This advanced challenge involves a lot of research and long production chains. In the end, you should be able to produce processing units from nothing but raw ingredients.

f) Rocket Silo (2 points) The victory condition of the Factorio game is to build a rocket for exploring space. The victory condition for this challenge is to craft the rocket silo that we need for that.

g) Spidertron (2 points) Can you figure out how to craft the mighty spidertron?

Exercise 5 — Bonus Challenge: Build Order Optimization

Finding a build order that is valid is one thing. The real challenge is to find a build order that minimizes the number of game ticks to the valid solution. Like every year, we will host a fierce competition on which team can compute the best build order.