TUSS Challenge

This document describes the setup and todo's for the Train Unit Shunting and Servicing (TUSS) challenge. If any questions remain, please contact J.G.M.vanderLinden@tudelft.nl

Table of Contents

Required/recommended Knowledge

Required

Python	C++	Cmake	Docker
Pybind11			

Recommended

Django	Doxygen	Google protobuf	

General description of the setup

The TUSS challenge consists of three main components:

Short name	Description	Git repository
TORS	The TORS source code	https://github.com/AlgTUDelft/cTORS, newest version on https://gitlab.tudelft.nl/jgmvanderlinde/ctors.
tussc-web	The TUSS challenge server	https://gitlab.tudelft.nl/jgmvanderlinde/tussc-web (private, available on request)
tussc-chal	The TUSS challenge	https://gitlab.tudelft.nl/jgmvanderlinde/tussc-chal (private, should stay private, available on request)

The TUSS problem is explained on a high level in the description of the tussc-chal (see tussc-chal/templates)

1. The TORS source code

The TORS source code is now published at https://github.com/AlgTUDelft/cTORS, but the newest version is published at https://gitlab.tudelft.nl/jgmvanderlinde/ctors. Documentation for this repository can be found at https://algtudelft.github.io/cTORS/ .

The repository contains code for automatic build on Gitlab. This is not configured yet for Github.

Compilation

This code is compiled using cmake. See the readme for how. It is tested on a Ubuntu 20.04 with GCC 9.3 as compiler. It requires the C++17 standard. The code also successfully compiles by using the Dockerfile in the repository. The current setup, as far as I know, does not work out of the box on windows.

Documentation

The documentation for cTORS is written in the doxygen syntax in the C++ files. See the readme

how you can use this documentation to generate a header file for the python module.cpp. In this way (almost) all of the c++ documentation is exported to the python package.

Structure

Folder	Description
cTORS	The C++ source code
cTORSTest	The tests for the C++ code (not many at the moment)
pyTORS	The pybind11 binding from c++ to python (compiles to the pyTORS python package)
TORS	The python code that uses the pyTORS python package to run and evaluate agents.
TORS/visualizer	A visualizer for TORS in the browser using flask and svgwrite. This visualizer can be used to manually solve scenario's, or to replay solved/failed scenario's.

2. The TUSS challenge server

The challenge server is a fork of the https://github.com/Cloud-CV/EvalAI repository. See the commit history for the changes. Check https://evalai.readthedocs.io/en/latest/ for the documentation of the evalai server.

After installing docker and docker-compose, you can run the development server locally by running `docker-compose up --build`

This runs the server at `localhost:8888`.

Here you can login with username: host, password: password, or username: participant, password: password

It also has an admin page at `localhost:8000/admin`

Here you can login with username: admin, password: password

Main changes:

1. The worker image now uses docker to run its submission. The submission should be .zip file containing a Dockerfile with its context. Note however, that the challenge config file (in tussc-chal/challenge_config.yaml) has "is_docker_based" set to False. This last setting would allow to use the EvalAI CLI (see https://cli.eval.ai/) to upload docker images. But this assumes that Amazon AWS is used as backend. Instead, the code is changed so that the docker image is build and run locally. See also the file tussc-chal/evaluation_script/main.py.
2. The original server allowed users to create a host team and participant teams. Because this server is intended to host only one challenge, this is disallowed. Only the admin/staff user can create/change the challenge. You can assign this privilige by going to the admin site change the host user's property 'is_staff'.

Likely questions/issues:

- After uploading a challenge (which needs to be done after the server is built for the first

time), the challenge does not appear in the challenge list. The challenge first needs to be approved by the admin. So, login in the admin panel (localhost:8000) as admin and mark the challenge as admin approved.

- After approving the challenge, you need to restart the worker:

  ```
  docker-compose restart worker
  ```

3. The TUSS challenge

The TUSS server is a fork of the EvalAI server, which is designed to run multiple challenges. Challenges are uploaded to the server. Therefore the specific challenge code is in a different repository than the server.

It contains the following folders and files:

annotations	Contains json files that describe the different phases of a challenge. In this challenge these files just refer to a episode config file in the evaluation_script folder
evaluation_script	Contains the evaluation script and the episode configuration for the different phases. It also contains a data folder for the scenario instances that are not to be shared with competition participants. Note: the data folder is now ignored by the .gitignore file. It can be obtained by request
solution_template	The Dockerfile contains a sample solution. It uses the tors-base image as its base. Participants can extend this file to install extra applications, copy files, install python packages etc. It should update the TORS/agent.json file to refer to the submitted planner. The docker folder contains the Dockerfile for the tors-base image. This image can be used as a base from which a solution can be made. The test_locally.py file can be used to test the submission locally. It can be run by using python test_locally.py -i Dockerfile The run.sh can be run to zip a solution. This zip-file can then be submitted on the website. Note: The Dockerfile contains a line to either 1) copy the TORS folder, or 2) to get it from gitlab. If you want to test local untested code, use the first option. The TORS folder must be in the same folder as the Dockerfile
templates	Contains html files that describe the challenge. This html is uploaded and stored in the server's database. Once the competition is live, change the database contents to update the website's appearance
Run.sh	Run this file to gather all the challenge files into a zip file. This zip file can then be uploaded on the server to create a new challenge.
Others	The other files and folders are not important are not being used.

New phase

In order to add a new phase to the challenge:

1. Add a new phase to the challenge_config.yaml
2. Add a new dataset_split to the to the challenge_config.yaml
3. Add a new challenge_phase_split to the challenge_config.yaml
4. Add a file to the annotations folder (point to this file in the challenge_config.yaml
5. Add a episode config file to the evaluation_script folder
6. Add data for the phase to the evaluation_script/data folder
7. Add a phase description in the templates folder

(The idea is that a phase can consist of several dataset splits. But in this competition, for now, every phase has only one dataset split)

Uploading updates

When you update the challenge, you need to upload these changes to the server. It is not desired to remove the whole challenge on the server and to upload a new challenge. Instead do this:

1. Make changes to the scripts/annotation files
2. Run run.sh in the tussc-chal root directory
3. In the server's database, find the address of the challenge's data (it is stored on the server in a .zip file with a random name)
4. Replace this .zip file with the zip file generated by the run.sh script
5. Restart the worker (run the command from the tussc-web directory):

```
docker-compose stop worker
docker-compose up -d --force-recreate --no-deps --build worker
```

In case you only update the challenge_configuration.yaml or the files in the templates folder, you can directly apply these changes in the database.

List of TODO's

1. Deploy the server (must)

The current server can be run locally by executing `docker-compose up` in the repository folder (note, the folder name must be `evalai`). This starts the development server. A separate docker-compose file is available to start the production server. Also, in the folder `scripts/deployment` there are some scripts to run for deployment. I did not completely figure out how to do this. Some issues:

1. In several settings files the host name is set to evalai.cloudcv.org (or something similar). All these need to be updated to the host name of the challenge website. Use the following command to find all the files that need to be changed:

   ```
   grep –r "cloudcv.org" *
   ```

   Suggestion: replace these hard-coded references to the hostname url with an environment variable

2. The whole setup of EvalAI is to use Amazon AWS as a backend for the docker execution. Our initial design was to ignore this. This now works in the dev environment, but in the production environment there are still many references to the amazon server. These need to be removed.

   Suggestion: use the dev setup/settings as the starting point and change these files rather than changing all the production files

3. Change the default passwords
4. Change the url's of the static images. The tussc-chal/templates files refer to images, but their addresses are still set to localhost
5. Setup and configure a e-mail server so that the web server can send e-mail addresses (for example for account confirmation)
6. Forward the local ports to the docker ports, and allow only the necessary ports to be accessed remotely
7. (Optional) merge the newest commits of the EvalAI repository into the tussc-web repository.

(It is suggested to do this part together with someone who has experience with Django. Otherwise it will cost a lot of time to figure out how all of the server setup works)

2. Publish challenge solution template (must)

The tussc-chal repository now is private, and should stay private because it contains the evaluation script for the challenge.

1. The subfolder solution-template can be published as a separate public repository that can be forked by others to begin their solution submission.

3. Security checks against fraud or abuse (must/could)

The tusss-web server allows for code to be run. This can be quite risky, both from a security perspective and from a competition perspective.

1. (Must) Make sure that the docker is run without any network privileges
2. (Could) Currently the evaluation script checks that none of the TORS files are changed (by checking their checksum). Other files could also be added to this checklist of files that are not allowed to be changed. E.g. the python files are checked, but they are already installed. In theory the competitor could uninstall the pyTORS python package, replace it with its own so that it always returns successfully immediately. Every system of course can be hacked, so winning solutions should be checked manually. But at least some more files could be checked.

4. Write the legal terms of the competition (must)

1. The legal terms and conditions for participating in the condition still need to be written

5. Cleanup submission docker container and images (should)

The worker script (tussc-chal/evaluation_script/main.py) now contains a command to remove the docker image and container after it has been run, but quite often my local hard disk got full because of all of these build images and containers. Apparently some of them are not removed. Probable cause, if the script fails, maybe the clean-up code is not reached. Or it is some other failure.

1. Try to remove all the docker files in a try-finally clause
2. Search for other issues
3. If nothing works, write a clean-up script that removes those unused containers and images, but take care that you don’t remove the server images and containers!

6. Provide access to and licenses for solvers (should)

Currently the tors-base docker image comes without cplex, gurobi or any of these installed. Specifically people from OR might want to use these for their solutions.

1. We could install these already in (one of) the base images
2. The next question is how to deal with licenses. On the TU Delft, technically, the university license could be used. The question is if this is also legally allowed.

7. Faster/easier TORS compilation/installation (could)

1. Find a work around for google protobuf. Now compiling of cTORS requires google protobuf to be installed, which takes a long time. Maybe installing of protobuf could be skipped and only include the header files
2. Provide cTORS as a python package that can be installed using pip/conda instead of having to compile it locally

8. Publish the docker images (could)

The challenge solution template now contains a Dockerfile that can be build and used as a starting point for the docker submissions. The user now first needs to build this image and then this image can be used as starting point for their own submission.

1. Instead you can publish the tors-base image (tuss-challenge/solution-template/Dockerfile) as a docker image on a docker repository
2. Also, several versions of tors-base could be offered. The current one is based on the tensorflow image, to kick start reinforcement learning solutions. For many solution methods, this is not necessary and a much smaller image (e.g. gcc 9.3) would be sufficient

9. TORS baseline solutions (could)

The idea was to provide some baseline solutions with the simulator. Those are not there yet. Currently the code includes a (1) random solver, (2) a greedy solver, and (3) a very simple RL solution based on stable_baselines. The greedy solver can solve 100% of instances with one shunting unit, but with multiple units, completely ignores the other shunting units. The RL solution is a very simple attempt, and works for at most 3 trains.

1. Base line solutions could be included.

Future work

1. Next phases

The following table shows the planning for each of the phases. Yellow text means it is a new feature in comparison to the previous phase (in phase one it is compared with general multi-agent path finding). Underlined means it is not implemented yet.

Phase 1	**Phase 2**	**Phase 3**
Single trains	Combined trains	Combined trains
One type of service tasks	Multiple service tasks	Multiple service tasks
One service facility	Multiple facilities	Multiple facilities
Multiple trains per track	Multiple trains per track	Multiple trains per track
Trains pre-matched	Matching problem	Matching problem
		Employee track
		Uncertainty track
	Scheduled trains crossing the yard	Scheduled trains crossing the yard

2. Employees

The current TORS source already contains some references to employees but as of yet does nothing with employees.

3. Uncertainty

Disturbances still need to be added to the TORS source. There are some stubs already, but more thought needs to be put in to change TORS to allow for disturbances.

The most important question is how to deal with the main design point of TORS: event-action based design. TORS now operates with an event queue. The duration of a wait action is until the next event. Disturbances would mean

1. Extra unforeseen events are added to the event queue. The current though on how to implement this is to have two event queues. On in the state, visible for the agent; the other in the scenario object, not visible by the agent. (Instead of an event queue in the scenario object, there could also be a disturbance probability distribution in the scenario object)
2. The duration/timing of some actions/events becomes uncertain. The event queue now is a deterministic priority queue with the priority set by the time of the event. The queue could be re-implemented as a non-deterministic queue, with uncertain finish times for the events and actions (Action-Finish events). Wait actions then should also be implemented differently.

3. Make TORS fully compatible with the NS files (optional)

The NS uses google protobuf to describe their data files. The proto files included in TORS are not precisely the same, but slightly simplified. The NS proto files are available on request to Joris den Ouden (joris.denouden@ns.nl), but this may require you to sign a non-disclosure agreement.

Their HIP (local search) solver generates plans that uses the same format as is used by TORS. However, they do not presume precisely the same meaning. The precise definition of the NS HIP plan format is not known to me and further communication with NS should help to solve this.

An example difference:

- In the NS plans, an Exit or Arrive action is always described by both a Move action and a Arrive/Exit action, whereas TORS only outputs the Arrive/Exit action

Challenge Description

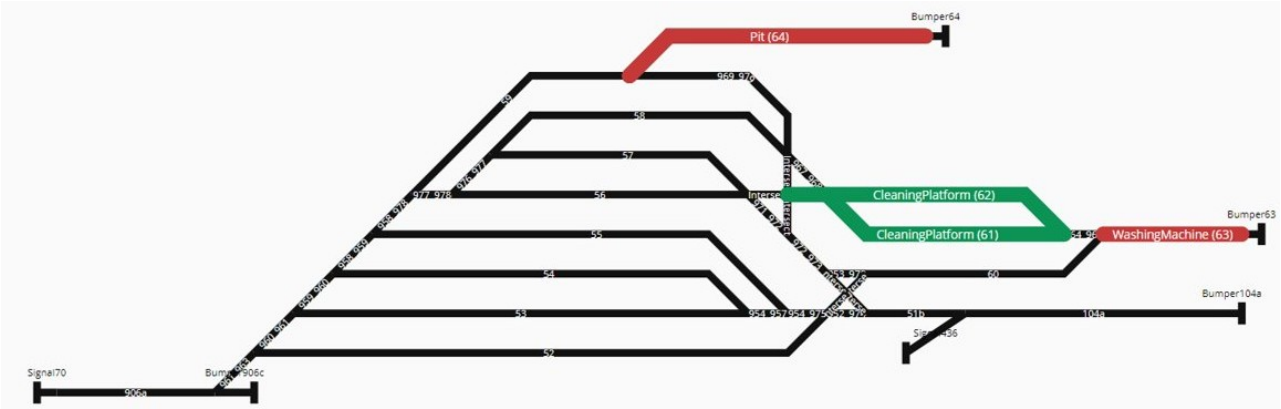
(This description can also be found in tussc-chal/templates)

Challenge motivation

With this competition we aim to cater to a wide range of researchers, from OR, TCS/Alg and AI; supporting also hybrid solutions, including for example reinforcement learning. Our goal is to create a competition that tests the robustness and flexibility of scheduling algorithms in an uncertain environment. In this competition the challenge domain is Train Unit Shunting and Servicing (TUSS).

The Train Unit Shunting and Servicing Domain

In between transportation services, trains are parked and serviced at shunting yards. The conflict-free routing of trains to and on these yards and the scheduling of service and maintenance tasks is known as the train unit shunting and service (TUSS) problem. Efficient use of the capacity of these yards is becoming increasingly important, because of increasing numbers of trains without proportional extensions of the yards. The objective is to position all the trains at the right position, and to get as much of the (optional) maintenance tasks done in limited run time.



Detailed overview of TORS

Location

The shunting yard is modeled as consisting of several tracks. Every track is connected to a number of tracks on the A-side and on the B-side. Each track has a type: Bumper, Railroad, or Switch. A *bumper* is the end of a track and is only connected on the A-side to one other track. A *railroad* is a track that is connected to one track at both the A- and B-side. Railroads have a length, and trains may be parked on railroads (if allowed). There are several types of switches. The *switch* is a track splitting into two tracks. An *English Switch* connects both A-side tracks to both B-side tracks. A *Half English Switch* connects A-side-1 to both B-sides, but A-side-2 is connected only to B-side-2. An *Intersection* connects four tracks, but A-side-1 is only connected with B-side-2, and A-side-2 is only connected to B-side-1.

The tracks also have a number of attributes (which are mostly ignored in phase 1), e.g. is_electrified. The important ones for phase 1 are:

- length: this number describes the length of the track in meters (only Railroad tracks) and determines the amount of trains that can be parked on a track at one moment in time.
- saw_movement_allowed: only if this attribute is true, can setback operations be performed on this track.
- parking_allowed: only if this attribute is true, is parking allowed on this track.

Trains are directed. Therefore a train's position cannot only be described by its current track, but also by its previous track. For this same reason, when you run, for example, the location.get_shortest_path function, you need to provide four tracks as parameters: (from_previous, from_track, to_previous, to_track).

When a train arrives, the incoming object has a parking_track attribute and a side_track attribute. The latter describes the "bumper" from which the shunting unit will enter the yard. Once it has arrived, this side track becomes the previous track of the shunting unit. For the outgoing shunting unit, the side_track describes the "bumper" over which the shunting unit will depart from the yard.

Shunting units

In TORS, the most important entities are *shunting units*. A shunting unit is a combination of *one or more trains* that is shunted as one unit. In this terminology (unlike what you might expect) a train is considered to be an atomic unit (even though a train exists of several carriages), i.e., in TORS a train cannot be split. The number of trains and shunting units is therefore not necessarily the same. Note also that this means that trains are atomic and persistent, whereas shunting units are transient!

Every train has a unique id. The shunting units also have id's, which are unique on a given moment in time, but over time the same id can be used for different shunting units. For example, a scenario may have an incoming shunting unit with id 33333, consisting of two trains with ids 2401 and 2402, and an outgoing shunting unit also with id 33333, but with one train with id 2401.

In the first phase of the competition, only shunting units consisting of one train are considered, so this can still be ignored. But for later phases the distinction between a shunting unit and a train will become important.

Also, in the second phase, the matching problem will be added to the challenge. This means that outgoing goals will not be specified by train ids, but by train types, e.g. a shunting unit with id 11111 should leave from track 1 at time 800 consisting of two trains of type SLT4 (instead of two trains with id 2401 and 2402).

Every train has a type. In phase 1, the train type influences the problem in two ways:

- The train type determines the length of the setback action. Therefore the location.get_shortest_path function also requires the train type as a parameter.
- The train type determines the length of a shunting unit. This is important, because fewer longer trains can fit on one track than short trains.

In summary:

- Trains are atomic and persistent. Shunting units are transient and can be split and joined together.
- Train ids are unique. Shunting unit ids are only unique on that moment in time, but may be reused later in time.

- Incoming and outgoing goals are defined on shunting units.
- Maintenance tasks are defined for trains.

Shunting units ordering

Shunting units may consist of several train units, and several shunting units may be parked on one track. So how, is the order determined? Track occupations are stored in a list that should be read from A-side to B-side. So if track 1 is occupied by a list of shunting units: SU1 and SU2, the track will look like: A < --- SU1 - SU2 ---- > B.

Every shunting unit consists of a list train_units of trains. One of these trains is the *front train*. The function state.get_train_units_in_order gets the order of the train units. If train_units[0] is the front train unit, the list is returned, otherwise the reverse of the list is returned. So a SU1 with a train_units list of [2401, 2402] with 2401 as the front unit looks like this: SU1 (2402 - 2401>).

The direction of the train on the track is determined by its previous track. The state.print_state_info function prints the occupations for every track. If the previous track is on the B-side of the track, the train units (ordered by state.get_train_units_in_order) are reversed. The trains are then printed last to first. (See state.get_train_units_in_order for the full code.)

As an example, take the shunting unit suA that is parked on track 1 as follows: A < ---- suA (T3 - T2 - T1>) --- > B. If this shunting unit is split on position 1, the first train (T1) would become a separate shunting unit, resulting in this situation: A < ---- suB (T3 - T2>) - suA (T1>) --- > B.

TORS simulation loop

TORS is a state-event-action based simulator and operates as follows:

1. An initial state is generated from the scenario description.
2. While the scenario is not finished (i.e. there are remaining events or the scenario end time is not reached yet):
    1. Execute all events that are triggered on this timestamp.
    2. If no action is required (i.e. all shunting units have an active action, go back to 2.1)
    3. Ask the planner for an action.
        1. If the planner returns no action, check if indeed no action is possible. If correct, the scenario is failed. Otherwise the planner is planner is penalized for making an invalid action.
        2. If the planner returns an action, execute it. (if invalid, stop the scenario and penalize the planner.)
3. Assign a reward to the planner, if the scenario is solved successfully.

Actions

Every time an action is required, the planner is asked to give one. The TORS simulator has two different classes for actions: Action and SimpleAction. In general, Action is used for the internal representation of actions. SimpleAction is used to communicate with external parties, such as the planner. The planner can either:

- Ask the engine for a list of all the valid actions. This will return a list of Action instances. The planner then can choose one from this list.
- Or, the planner can choose an action by itself, by constructing a SimpleAction and returning

this.

(Note that in the pyTORS module the Action class and its derivatives do not have constructor methods, and the SimpleAction with its derivatives do have constructor methods.)

Also note that even though an action is valid it does not always end up in a valid state (an invalid state is a state in which an action is required, but no action is available). For example a shunting is moved to another track, but while it moves it blocks all tracks that another shunting unit needs (and this other unit is not allowed to park).

The following actions are available:

- Arrive(inc): Let the shunting unit described by the Incoming instance inc arrive.
- Exit(su, out): Exit the shunting unit su, according to the outgoing goal described by out. Note that you need to specify which shunting unit you want to exit. This is because of the matching problem, where out does not describe a uniquely identifiable shunting unit.
- BeginMove(su): Begin a move operation for shunting unit su.
- EndMove(su): End a move operation for the shunting unit su. This means park the shunting unit.
- Move(su, to_track): Move shunting unit su to track to_track.
- Setback(su): Setback the shunting unit (that is, change the unit's direction).
- Service(su, task, train, facility): Execute the specified service task on the train in the shunting unit su at the given facility.
- Split(su, split_index): Split the shunting unit su at the specified index. TU3 - TU2 - TU1> with split index 2 is split into TU3> and TU2 - TU1>.
- Combine(su1, su2): Combine the two adjacent shunting units su1 and su2 into one unit.
- Wait(su): Let the shunting unit su wait until the next event.

Move actions and duration

When the simulator is asked to generate actions, it will generate move actions by checking for full origin-destination paths (without setbacks). When such paths are available, they are changed into step by step move actions. Such moves are always from one railroad track to a (indirect) neighboring rail track. Here one could consider the railroad tracks to be the nodes of a graph, and all the switches to be the edges. The is done to make the action space smaller.

The duration of move actions is determined as follows: the duration of a move action is #movement_constant + #railroad_coefficient * #number_of_railroad_tracks + #switch_coefficient * #number_of_switches.

Business rules

To check an action's validity, a number of business rules are checked. This table shows which business rules are checked. See the source documentation for the full details.

Name	Category	Description
end_correct_order_on_track_rule	arrival_departure	Rule that verifies that shunting units which stay in the shunting yard after the scheduling period will be located in the right order on their track.
in_correct_time_rule	arrival_departure	Rule that verifies that shunting units that are arriving, arrive at the correct time. Note: shunting units will never arrive too early, so this rule only

Name	Category	Description
		checks if a shunting unit arrives too late.
out_correct_order_rule	arrival_departure	Rule that verifies that leaving shunting units have their train units in the correct order when they leave the shunting yard.
out_correct_time_rule	arrival_departure	Rule that verifies that leaving shunting units leave at the correct time.
out_correct_track_rule	arrival_departure	Rule that verifies that leaving shunting units leave over the correct tracks.
blocked_first_track_rule	track_occupation	Rule that verifies that shunting units, upon starting a movement, are not blocked on exit by other shunting units on their current track.
blocked_track_rule	track_occupation	Rule that verifies that moving shunting units are not blocked by other shunting units.
length_track_rule	track_occupation	Rule that verifies that shunting units on a single track do not take up more space than available on that track.
single_move_track_rule	track_occupation	Rule that verifies that at most one shunting unit can use a piece of track at a given time.
electric_track_rule	parking	Rule that verifies that shunting units which need electricity park only on electrified tracks.
legal_on_parking_track_rule	parking	Rule that verifies that parked shunting units are on a track where parking is allowed.
legal_on_setback_track_rule	parking	Rule that verifies if a shunting unit is parked on a track where setback is allowed.
electric_move_rule	shunting	Rule that verifies that shunting units which need electricity park only on electrified tracks.
setback_once_rule	shunting	Rule that verifies that a setback action is not performed on a shunting unit which is already in a neutral state. A shunting unit is in a neutral state if a setback or service action is performed.
setback_track_rule	shunting	Rule that verifies that performing a setback action on a shunting unit is allowed on the track where the shunting unit is at.
available_facility_rule	facility	Rule that verifies that tasks assigned to a facility are only executed when that facility is available.
capacity_facility_rule	facility	Rule that verifies that no more tasks are executed at a facility than the facility can handle.
disabled_facility_rule	facility	Rule that verifies that no tasks are assigned to facilities which are disabled by a disturbance.
correct_facility_rule	service_tasks	Rule that verifies that service tasks are executed at the correct facility.
mandatory_service_task_rule	service_tasks	Rule that verifies that all required service tasks are performed before a shunting unit leaves the shunting yard.
optional_service_task_rule	service_tasks	Rule that verifies that all optional service tasks are performed before a shunting unit leaves the shunting yard.

Name	Category	Description
understaffed_rule	service_tasks	Rule that verifies that all tasks have enough employees assigned, with the right skills, such that the task will have all of its required skills available.
order_preserve_rule	combine_and_split	Rule that verifies that combining or splitting shunting units does not change the order of train units on a track.
park_combine_split_rule	combine_and_split	Rule that verifies that combine and split actions on shunting units are only performed on tracks where parking is allowed.
setback_combine_split_rule	combine_and_split	Rule that verifies that combine and split actions on shunting units are only performed on tracks where setback is allowed.

TORS configuration

The configuration of TORS is divided over several configuration files.

Episode.json

The episode.json in the cTORS/TORS folder describes the evaluation session. The file should be hidden for the competition participants. It contains the following settings:

- data folder: the folder where the data for the session is stored
- scenario: the scenario file (or folder, in case multiple scenario's are used)
- verbose: the level of the verbosity of the TORS manager and simulator (python code). If you want to run cTORS with verbosity, compile it with the debug flag.
- n_runs: The number of runs for every amount of trains. (The evaluation first runs with one train, then with two, etc… until max_trains. For every amount of trains, it runs n_runs scenario's.
- max_trains: The generated scenario's contain this maximum number of trains. The evaluation is performed starting with one train until max_trains. The evaluation is stopped early, if for n trains not a single run is ended successfully.
- time_limit: in seconds. The total run time available for the planner. Note that this is the total run time for all runs, not for each individual run. The planner should solve as many instances in the given time. Only the time spent in the planner is measured.
- generator: describes the generator used for scenario generator
- generator/class: The python class used for scenario generation
- generator/n_workers: <this setting is currently not used> number of employees to be generated.
- generator/n_disturbances: <this setting is currently not used> number of disturbances to be generated.
- generator/match_outgoing_trains: boolean. If true outgoing trains are matched (required for phase 1). If false, the outgoing train ids are set to "****" (any train id possible)

- generator/max_length: The maximum length of shunting units (number of train units). For phase 1 this should be set to 1.
- “generator_class_name”: extra configuration options for the specific class of generator (dictionary)

Agent.json

The agent.json in the cTORS/TORS folder describes the planner.It contains the following settings:

- class: The class name of the planner agent (that is, the python import path for the class)
- seed: the seed to be used for the random instances
- verbose: the level of the verbosity of the agent
- “agent_class_name”: extra configuration options for the specific class of agent (dictionary)

data_folder/config.json

In the data folder specified by episode.json there should be a config.json, a location.json (and optionally a vis_config.json, if you want to use the visualizer).

The config.json contains the following settings:

- business_rules: configuration for the business rules
- business_rules/categories: short cuts to disable whole categories of rules (currently ignored)
- business_rules/rules: per rule a dictionary with parameters:
  - on: boolean to turn the rule on or off
  - soft: (currently ignored) boolean to turn the constraint into a soft constraint
  - category: (currently ignored)
  - parameters: (currently ignored) extra parameters for the business rule
  - priority: (currently ignored)
  - status: (currently ignored)
- actions: settings for every action type:
  - parameters: a set of extra parameters (e.g. for setback how to calculate the move duration)
  - on: boolean to turn this action on or off (for phase 1 split and combine are off)
  - status: (currently ignored)
- employees: (currently ignored) whether to include employees in the problem
- verbose: (currently ignored)
- br_break_on_fail: (currently ignored)
- routing: (currently ignored)
- distance_matrix: (currently ignored)

data_folder/location.json

The same folder also contains the location.json config file which describes the location. All the tracks and facilities are described in a clear way. It also contains the following settings:

- movementConstant
- movementTrackCoefficient
- MovementSwitchCoefficient

These three are used to calculate the duration of move actions.

Submission guidelines

Follow the following steps to upload your solution:

1. Download the solution template from <solution_template_git_url>.
2. Write your agent in the agent folder: agent/my_agent.py.
3. Write your agent's config file: agent/my_agent_config.json.
4. Update the Dockerfile so that it copies all your files into the docker image.
5. Update the Dockerfile so that it will install all your required packages.
6. Build the tors-base image (Dockerfile is in the docker subfolder)
7. Run test_locally.py to see if your solution is able to run successfully locally.
8. Run run.sh to bundle your files into a .zip file.
9. Upload the .zip file to the server.