Bus Network Optimization and Planning Interface (BNOPI)

Overview

Bus Network Optimization and Planning Interface (BNOPI) is a software designed to aid visualization and interaction with algorithms for the Urban Transit Routing Problem (UTRP) and related problems. BNOPI makes the assumption that the algorithmic network design process can be represented with a **dependency graph**, an example of which is shown below.

The role of BNOPI is to

- control execution of the dependency graph
- visualize the results of intermediate stages on a geographical map embedding
- allow users to make changes

In the dependency graph, each node represents an **algorithm**. Algorithms are accessed by BNOPI using a metadata file, which contains information about the command-line parameters, input and output file formats, and a shell script to launch the algorithm.

To transfer data between nodes, BNOPI uses **stage formats**. A stage format refers to the structure of some input, output, or intermediary file. The files themselves are referred to as **stage instances**, or instances of the stage format. An example of a stage format is `STOPS`, instances of which contain data about bus stops in a standardized JSON format. Each edge in a project's dependency graph represents the flow of a stage instance from one node to the next.

On a node, a **breakpoint** may be placed. This indicates that the resulting stage instances of that node should be visualized, and the user should be able to edit them before proceeding.

To accomplish this, **display frameworks** and **editing frameworks** can be defined on a stage format. The display framework is a JavaScript function that takes an instance of the stage format and returns a list of items for BNOPI to display, such as bus stops or routes. BNOPI allows the user to edit these items in the GUI, and the editing framework makes the reverse conversion of these items back to a stage instance.

How to install

See /README.md

Example Project Walkthrough

BNOPI provides some standard stage formats and algorithm implementations. Open the example project test_project in BNOPI and observe the dependency graph.

The dependency graph's execution begins by downloading information about bus stops and roads from OpenStreetMap, then creating a "stop connection graph". This representation of the road network represents a connection between 2 bus stops as a single directed edge, weighted on the road distance between them. We then use a genetic algorithm to generate a fitting route network.

Preliminary steps

In order to run the dependency graph you will need to compile the genetic algorithm, written in C++. Find the metadata file for the genetic algorithm (located in /bnopi-algs/), referenced within which is the path to the executable which needs to be compiled. Navigate to that folder and follow the compilation instructions in the README.

The other algorithms used in this project are implemented in Python 3, and also referenced from /bnopi-algs/. You may need to install some modules for these to work.

Project directory structure

The main entry point for the project is `info.json`, located in the top level of the project. This contains information about the project and links to other files and folders. Projects are structured so that all the project data is referenced from the *info.json* file, and there are no other fixed file names. Below shows the contents of the info.json file.

```
{
    "title": "Test Project",
    "description": "Project used for testing.",
    "dependencyGraph": "test_dependency_graph.dg.json",
    "stageInstances": ["stage_instances/"]
}
```

`dependencyGraph` references a *.dg.json* file containing information about the structure of the nodes in the project. `stageInstances` contains a list of directories within which BNOPI will look for stage instance metadata (*.stg.json*) files, which are used to keep track of generated stage instances. Generally the stage instances are stored in the same folder alongside the *.stg.json* files.

Dependency Graph

The GEN_STOP_CONNECTION_GRAPH algorithm has an algorithm metadata (*.alg.json*) file, which can be found here. Inside the BNOPI interface, clicking on the cog icon in the GEN_STOP_CONNECTION_GRAPH node on the dependency graph allows us to view some of this information in the Properties tab to the right. You can edit the values of the paremeters as well as the title and description of the node. Note that this does not change the original *.alg.json* file; the changes occur in the dependency graph file *.dg.json*.

Running the project

After these steps, you should be able to run the project. Click the play button to execute until the first breakpoint. The `import-stops-from-osm` and `import-roads-from-osm` algorithms should run. You will be able to edit the bus stops that were downloaded from OpenStreetMap in the interface.

The “create bus stop” and “delete bus stop” tools in the left-hand-side toolbar can be used to edit the STOPS stage instance. You can also see a list of the current bus stops in the “Map info” tab on the right-hand-side sidebar under the “List of stops” heading. From there you can click the trash can icon to delete the stop or the crayon icon to edit info such as the stop’s name.

Continuing the execution, we generate a route network using a genetic algorithm.

Documentation

More in-depth documentation on the project structure can be found below.

- Projects
- Stage instances
- Stage formats
- Algorithms
- Dependency graphs

Projects

A BNOPI project consists of some metadata in a file named `info.json`, the project dependency graph, and the stage instances generated by it.

The project directory structure, as is generated when creating a new project in the UI, is the following:

```
.
\-- my_project/
    |-- info.json
    |-- dependency_graph.dg.json
    \-- stage_instances/
        \-- ...
```

The only part of the above that is fixed is `info.json` in the top level of the directory. The paths to the other components of a project are specified within it.

info.json structure

A project's *info.json* file contains information about the project and links to the other project components. It is a JSON Object containing the following properties:

- `title: String` - Project title. Typically the same or similar to the directory name.
- `description: String` - Description of the project.
- `dependencyGraph: String` - Relative or absolute (relative preferred) path to the project's dependency graph (*.dg.json*) file.
- `stageInstances: Array[String]` - Relative or absolute (relative preferred) paths to directories containing stage instance metadata (*.stg.json*) files to be displayed in the project's stage tracker.

Stage Instances

A **stage instance** is a file that is used as input to, or generated as output from, an **algorithm**. To keep track of stage instances, for each stage instance BNOPI generates a **stage instance metadata file**, with the file extension *.stg.json*.

BNOPI searches for *.stg.json* files in directories referenced from the project's *info.json* file, but if the project folder was generated using the user interface then this will always be a folder named *stage_instances* inside the project folder.

Parents and Siblings

Each *.stg.json* file contains JSON Arrays `parentStageInstances` and `siblingStageInstances`.

The **parents** of a stage instance are instances that were used as input to the algorithm that generated it. If the stage instance was instead generated by a user edit in the UI, then the parents are the original instances that were used in the display framework, including the requirement instances. `parentStageInstances` contains paths to the *.stg.json* files of the parents in the order that they appeared in the algorithm metadata file, or, in the case of user edit, the primary instance followed by the requirement instances in the order they appeared in the *fmt.js* file.

The **siblings** of a stage instance are other instances that were generated by the same run of an algorithm. For example, if an algorithm generates 2 stage instances, then each instance is a sibling of the other (but not of themselves). If the stage instance was generated by a user edit in the UI, then the siblings are any other instance that was generated by the editing framework. This will not necessary include new copies of all the

parents, as editing frameworks are not required to create such copies for each of the primary instance and requirements.

Stage instance metadata (*.stg.json*) file structure

The *.stg.json* file contains a JSON Object with the following properties:

- `timeCreated: string` - The time at which the stage format was created, in the format produced by the JavaScript function `Date.toJSON()`
- `dependencyGraph: string` - Path to the dependency graph file (*.dg.json*) at the point where the stage instance was created. Relative or absolute paths are accepted, but relative paths are recommended as this allows projects to be more easily copied between devices.
- `format: string` - The id of the stage format of this instance
- `generatedBy: string` - The id of the algorithm that generated this stage instance. There are 2 reserved ids, USER_PRIMARY_EDIT and USER_REQUIREMENT_EDIT, which are used if the instance was generated by an editing framework. "PRIMARY"/"REQUIREMENT" refers to the role of this instance in the editing framework.
- `nodeInGraph` - Node in the dependency graph that generated the instance. If the instance was generated from a user edit, then `nodeInGraph` refers to the same node that created the original instance.
- `parentStageInstances: Array[string]` - Paths to the stage instance metadata (*.stg.json*) files of the parents (as described above). Paths relative to the location of this metadata file are preferred.
- `siblingStageInstances: Array[string]` - Paths to the stage instance metadata (*.stg.json*) files of the siblings (as described above). Paths relative to the location of this metadata file are preferred.
- `datafile: string` - Path to the actual stage instance. A path relative to the location of this metadata file is preferred.

Example

Below is the contents of a *.stg.json* file.

```
{
    "timeCreated": "2023-04-20T10:55:53.687Z",
    "dependencyGraph": "../test_dependency_graph.dg.json",
    "format": "ROUTES",
    "generatedBy": "USER_PRIMARY_EDIT",
    "nodeInGraph": null, /* currently unused */
    "parentStageInstances": [
        "routes.stg.json",
        "stop-connection-graph.stg.json",
        "stops.stg.json"
    ],
    "siblingStageInstances": [
        "stops_USER_REQUIREMENT_EDIT_1.stg.json"
    ],
    "datafile": "routes_USER_PRIMARY_EDIT_1.json"
}
```

The stage instance being referred to by `"datafile"` was created as the result of a user edit. It was the primary instance in the editing framework for ROUTES, shown by `"generatedBy": "USER_PRIMARY_EDIT"`. The editing framework took 3 instances in total, the primary instance (`"routes.stg.json"`), and 2 requirement instances (`"stop-connection-graph.stg.json"` and `"stops.stg.json"`). The editing framework made a new copy of the routes (this metadata file) and the stops, the latter of which is referenced by the relative path `"stops_USER_REQUIREMENT_EDIT_1.stg.json"`. The editing framework did not create a new version of the stop connection graph, as this does not appear in the siblings.

Stage Formats

Stage formats are used to give a type to the files (stage instances) that are transferred between nodes. Within a project's metadata files, a stage format is referenced by its **stage format id**, which is conventionally a string comprised of uppercase letters and underscores, such as STOP_CONNECTION_GRAPH.

For some stage formats, *.fmt.js* files have been written. These files, stored in the /bnopi-stage-fmts/ directory, tell BNOPI extra information about the stage format, including how to display and handle edits to an instance of this format in the UI. The rest of this section is concerned with the structure of *.fmt.js* and how BNOPI uses them.

Procedure for displaying a stage instance in the UI

The stage tracker displays a list of *.stg.json* files which contain some metadata and point to the actual stage instance. The user selects an item from this list, which will be referred to as the **primary instance**.

BNOPI considers the stage instance metadata (*.stg.json*) file of the selected primary instance. This contains the `format` property, which is the instance's stage format. If there is a *.fmt.js* for this stage in the directory /bnopi-stage-fmts/, then this will have been loaded when BNOPI is opened.

In some cases, opening a stage format has some extra **requirements**; these are other stage instances that are needed in combination with the primary stage instance in order to display useful information of the map. For example, displaying a ROUTE_NETWORK instance requires access to a STOP_CONNECTION_GRAPH instance, because the route network references edges in the stop connection graph. If there are requirements, the user will be prompted to select the requirement stage instances. BNOPI will try to guess the appropriate instances based on the **parents** and **siblings** of the primary instance and display the most likely instances first in the selection boxes.

BNOPI then reads the primary instance (referenced by the property `datafile` in the *.stg.json* file) and the requirement instances into `Buffer` objects. It calls the **display framework** of the stage format, defined in the *.fmt.js*. This is of the type:

```
(method) displayFramework(primaryInstance: BNOPIInstance, requirementInstances: BNOPIInstance[], stageF
    stops: BNOPIStop[];
    routes: BNOPIRoute[];
}
```

`BNOPIInstance`s contain the instance itself, the contents of the instance's *.stg.json* file, and the path to the *.stg.json* file. It is typed as:

```
type BNOPIInstance = {
    data: Buffer;
    metadata: InstanceMetadata;
    metadataFilePath: string;
}
```

The display framework function generates a list of `BNOPIStop`s and `BNOPIRoute`s from the input instances. For each `BNOPIStop` a bus stop marker is placed, and for each `BNOPIRoute` a polyline is drawn.

`BNOPIStop` represents a bus stop, and is typed as

```
type BNOPIStop = {
    lat: number;
    lon: number;
    id: number; // integer
    name: string | undefined;
    hidden_attrs: any;
    user_attrs: any;
```

```
}
```

`lat` and `lon` are required. If multiple stops have the same `id` then only the first is used and the rest are ignored by BNOPI. If any `id` is `null` or `undefined` then BNOPI assigns a new id that is different to all others. `hidden_attrs` stores any data needed by the editing framework to reconstruct the primary instance after editing, but is not displayed to the user. `user_attrs` contains information that is displayed to the user and can be edited by the user.

`BNOPIRoute` represents a directional route, typed as

```
type BNOPIRoute = {
    id: number; // integer
    name: string | undefined;
    links: {
        lat: number;
        lon: number;
    }[][];
    stops: number[]; // integer[]
    hidden_attrs: any;
    user_attrs: any;
}
```

where the `links` list is a list of links (path between 2 bus stops), each link containing geographical points that the route passes through, and `stops` is a list of stop ids, referencing `BNOPIStop`s.

Procedure for editing a stage instance

Inside the interface, once a stage instance has been loaded using the display framework, stops and routes may be created/edited/deleted, and any attribute in their `user_attrs` may be edited.

When the user saves changes to the stage instance, either by selecting another instance or manually selecting Save from the application menu, the **editing framework** for the stage is called. This is of the type:

```
(method) editingFramework(primaryInstance: BNOPIInstance, requirementInstances: BNOPIInstance[], stops:
    primaryData: (Buffer | null);
    requirementDatas: (Buffer | null)[] | null | undefined;
}
```

The return values, `primaryData` and `requirementDatas`, are new versions of the files to reflect the changes the user has made to the `stops` and `routes`. Values of `null` or `undefined` indicate to BNOPI that their instances have not been changed, and BNOPI will not write a new file.

Any stop or route that was added by the user will be given a new unique stop id or route id *before* the editing framework is called.

BNOPI then writes any new stage instances to disk (at locations decided by BNOPI) and creates accompanying *.stg.json* files.

.fmt.js structure

A *.fmt.js* file is a Node.js module that exports a class that extends the `StageFormat` class, defined in /main/js/stage_format_handler.js. It has the following properties:

- `name: String` - The name of the stage format
- `id: String` - An id by which the stage is referred to. Usually an uppercase string separated by underscores, e.g. `"STOP_CONNECTION_GRAPH"`.

- `requirements: String[]` - A list of stage format ids corresponding to the requirements of the display and editing frameworks.
- `description: String` - A helpful description of the stage format
- `fileExtension: String` - The preferred file extension for instances of this stage (without a dot before, i.e. just `"json"` for JSON files).
- `displayFramework` - A function used to convert the instances something displayable by BNOPI (see above)
- `editingFramework` - A function used to convert the BNOPI representation back to a stage instance (see above)

Algorithms

Algorithms are the building blocks of BNOPI projects. An algorithm is a script or executable which takes in 0 or more stage instances as input and produces 0 or more stage instances as output.

Example

In /bnopi-algs/ there is an **algorithm metadata file** (*.alg.json*) for an algorithm called GEN_STOP_CONNECTION_GRA which takes as input a STOPS and a ROUTES instance as well as some **parameters**, and outputs at STOP_CONNECTION_GRAPH instance.

Looking at the *alg.json* file directly, it contains a `name` and `description`, as well as information concerning the algorithm's inputs and outputs. Starting with one of the 2 types of input, this information consists of:

- The **parameters** - These are settings for the algorithm that can be set by the user before the algorithm is run. For GEN_STOP_CONNECTION_GRAPH the parameters are:

```
"params": [
    {
        "name": "Left-side drive",
        "type": "dropdown",
        "choices":["left", "right"],
        "default": "left",
        "help": "Specifies that road users should on the left. If omitted, it is assumed
         ↪  road users drive on the right.",
        "var": "DRIVE_ON_LEFT"
    },
    {
        "name": "Max search distance",
        "type": "posint",
        "default": 1000,
        "help": "Specifies the maximum distance to search for neighboring stops, in
         ↪  meters. Defaults to 1000.",
        "var": "MAX_SEARCH_DISTANCE"
    }
]
```

In the UI Properties tab, each parameter generates an input box for the user of the type `type`, which they can use to input the **value** of the parameter. Later, the values that the user inputs are referenced by the name in the `var` property.

- The **input stage instances**. This is how BNOPI passes data between nodes. We must specify the stage formats of the instances. For GEN_STOP_CONNECTION_GRAPH, the input stage formats are:

```
"inputStageFormats":[
    {
        "name": "Stops",
        "help": "The stops between which to make the stop connection graph edges",
        "var": "STOPS_FILELOC",
        "stage_format": "STOPS"
    },
    {
        "name": "Roads",
        "help": "Roads between the stops",
        "var": "ROADS_FILELOC",
        "stage_format": "ROADS"
    }
]
```

When the GEN_STOP_CONNECTION_GRAPH node is about to run, with the absence of a breakpoint on any parent node, BNOPI will automatically select instances of these formats generated by the parents. If, as is the case in the example project, there *is* a breakpoint, then the user will be allowed to edit the instances generated by previous nodes, and manually select which ones to use for the GEN_STOP_CONNECTION_GRAPH node. The file paths of the selected instances are later referenced by the name in the `var` property.

Algorithms only have 1 type of output:

- The **output stage instances**. In the *alg.json* file, the `outputStageFormats` refer to the formats of these outputs, in the same way as with the input stage instances:

```
"outputStageFormats":[
    {
        "name": "Stop connection graph",
        "help": "A json file containing the generated stops",
        "var": "OUTPUT_FILELOC",
        "stage_format": "STOP_CONNECTION_GRAPH"
    }
]
```

When the STOP_CONNECTION_GRAPH node runs, BNOPI automatically assigns a new file location for each output stage instance and generates a corresponding stage instance metadata (*.stg.json*) file.

- The final part of the *.alg.json* file is the **launch script**. This is resposible for processing the `vars` and launching the algorithm. Currently launch scripts can be written in Python 3 only. In this case the GEN_STOP_CONNECTION_GRAPH algorithm is written in Python 3 also. The launch script for GEN_STOP_CONNECTION_GRAPH is as follows:

```
#!/usr/bin/env python3
from os import getenv, system, name

# decide whether to include the --drive-on-left flag
dol_flag = ""
if getenv("DRIVE_ON_LEFT") == "left":
    dol_flag = "--drive-on-left"

if name == "nt": # windows. must specify interpreter
    system(
        f'py -3 ../algs/stop-connection-graph.py {dol_flag} -d
        ↪ {getenv("MAX_SEARCH_DISTANCE")} -o "{getenv("OUTPUT_FILELOC")}"
        ↪ "{getenv("STOPS_FILELOC")}" "{getenv("ROADS_FILELOC")}"')
```

```
else:
    # otherwise assume we can read the shebang
    system(
        f'../algs/stop-connection-graph.py {dol_flag} -d {getenv("MAX_SEARCH_DISTANCE")}
        ↪  -o "{getenv("OUTPUT_FILELOC")}" "{getenv("STOPS_FILELOC")}"
        ↪  "{getenv("ROADS_FILELOC")}"')
```

The launch script is run with the environment variables `DRIVE_ON_LEFT`, `MAX_SEARCH_DISTANCE`, `STOPS_FILELOC`, `ROADS_FILELOC`, and `OUTPUT_FILELOC` set as required. These variable names are those set from the `var` properties in the *alg.json* file. Additionally, BNOPI sets the environment variables `WORK_AREA_LAT`, `WORK_AREA_LON`, and `WORK_AREA_RADIUS` for all algorithms to specify the circular area centered on the coordinates within which the algorithms should operate. The launch script then substitutes these into a command to launch the algorithm.

.alg.json structure

A JSON object containing the following properties:

- `name: String` - A descriptive name of the algorithm.
- `id: String` - The id of the algorithm as it is referred to elsewhere in the project
- `description: String` - A helpful description of what the algorithm does
- `params: Array` - List of Objects containing about the algorithm's parameters. Each Object contains:
  - `name: String` - Name of the parameter
  - `type: String` - Type of the parameter. This affects the type of input box the user will be given in the UI. Currently implemented types are `"dropdown"` for a drowpdown selection input and `"posint"` for a positive integer.
  - `default: String|Number` - The default value for the parameter.
  - `help: String` - Helpful message adding extra information about the parameter to the user.
  - `var: String` - The environment variable which will hold the value of this parameter when the launch script is run.
  - `choices: Array[String] | undefined` - If the `type` is `"dropdown"`, then `choices` holds a list of the choices the user can select from.
- `inputStageFormats: Array` - List of Objects specifying the input instances for the algorithm. Each contains:
  - `name: String` - Name of the input
  - `help: String` - Helpful message describing the input
  - `var: String` - The environment variable which will hold the path to the location of the input instance when the launch script is run.
  - `stage_format: String` - The stage format id of this input.
- `outputStageFormats - Array` - List of Objects specifying the output instances for the algorithm. Each contains:
  - `name: String` - Name of the output
  - `help: String` - Helpful message describing the output
  - `var: String` - The environment variable which will hold the path to the location to where the output instance is to be written.
  - `stage_format: String` - The stage format id of this output.
- `launchScript: Object` - Object containing the different types of launch scripts implemented for this algorithm.
  - `python3: String` - Currently the only type of launch script supported by BNOPI. This contains a path to a Python 3 launch script for this algorithm.

Dependency Graphs

Backend Node Structure

A JSON object containing the following properties:

- `id: String` - the ID of the node as referred to elsewhere int he project
- `name: String` - the name of the stage as set in the frontend of the project
- `params: JSON` - the JSON object containing the name and value of the parameters as set in the metadata file. This will be described in detail later in this file
- `parents: Array[String]` - list containing the ID of the parent node(s) of this node
- `description: String` - the description as set in the frontend and metadata files
- `input_stage_format: JSON` - JSON object containing information about the input stages to this stage, contains the unique identifier of the stage. Will be explained in detail later
- `output_stage_format: JSON` - JSON object containing information about the output stages to this stage, contains the unique identifier of the stage. Will be explained in detail later

params Object Structure

- `choices: Array[String]` - Optional, contains the possible options if the param can only have predefined options
- `default: String` - contains the default value if none is assigned
- `help: String` - Defined in metadata file
- `setVal: String` - The value set for the parameter in the front-end
- `type: String` - The type of the value, for example dropdown or posint
- `var: String` - The name of the environment variable associated with the parameter, set in the metadata file

Stage Format Object Structure

- `help: String` - A string set in the metadata
- `name: String` - Name assigned to the stage
- `setStage: String` - The file containing the stage, used as a unique identifier for the stage
- `var: String` - The name of the environment variable associated with the stage, set in the metadata file