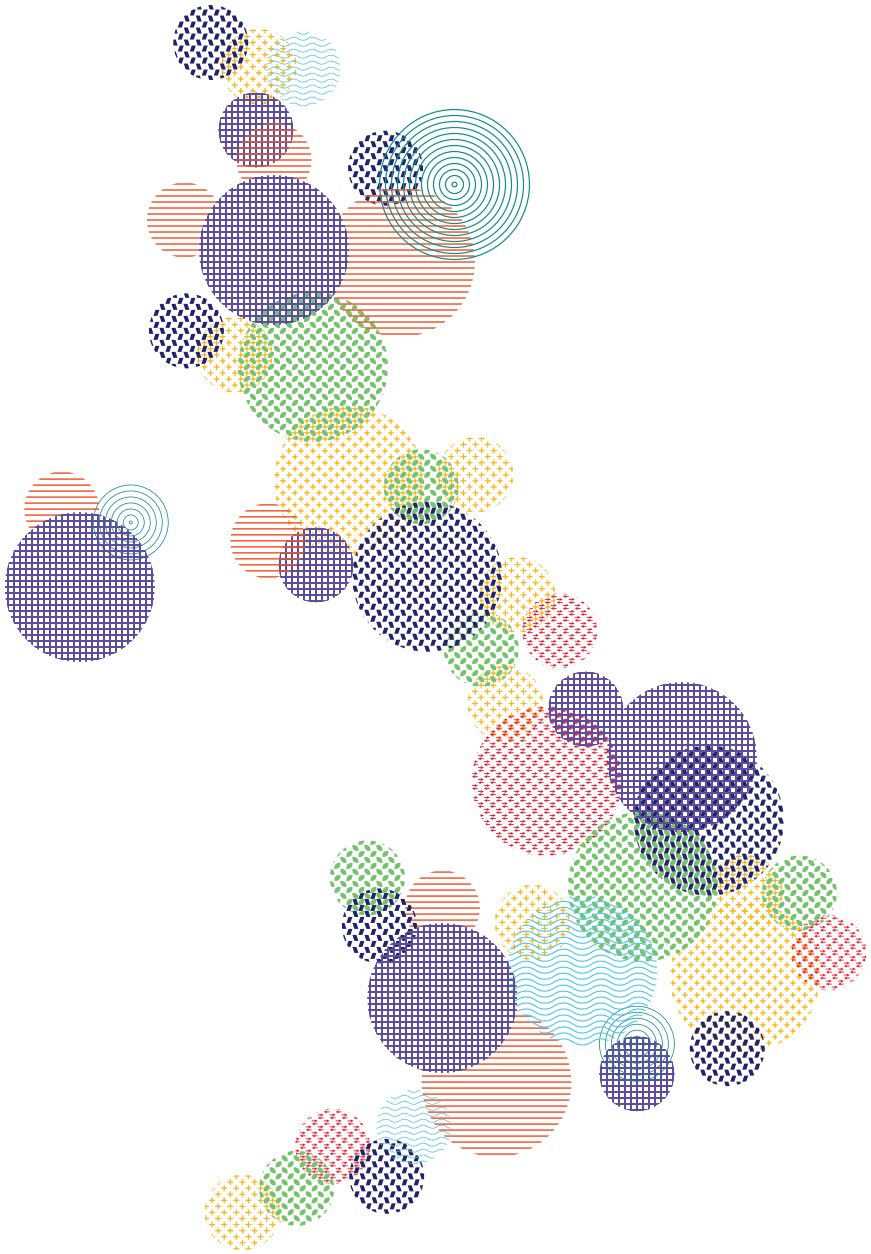
NATIONAL DIGITAL TWIN PROGRAMME

CReDo
Climate Resilience Demonstrator

Understanding infrastructure interdependencies and system impact

January 2022



Contents

Summary

In this report, we present the efforts done for understanding the impact of an inclement weather situation within a network. The network connectivity and the fault propagation is assessed for one network at a time; and then complemented with a propagation across networks. The modelling done for this demonstrator is a connectivity model which considers the dependencies between sites in a network.

The connectivity models assess the faults in each individual network—power, water and telecommunications.

1 Introduction

1.1 What is CReDo?

The Climate Resilience Demonstrator, CReDo, is a climate change adaptation digital twin demonstrator project developed by the National Digital Twin programme to improve resilience across infrastructure networks.

CReDo is a pioneering project to develop, for the first time in the UK, a digital twin across infrastructure networks to provide a practical example of how connected-data and greater access to the right information can improve climate adaptation and resilience. CReDo is the pilot project for the National Digital Twin programme demonstrating how it is possible to connect up datasets across organisations and deliver both private and public good.

Enabled by funding from UKRI, The University of Cambridge and Connected Places Catapult, CReDo looks specifically at the impact of extreme weather, in particular flooding, on energy, water and telecommunications networks. CReDo brings together asset datasets, flood datasets, asset failure models and a system impact model to provide insights into infrastructure interdependencies and how they would be impacted under future climate change flooding scenarios. The vision for the CReDo digital twin is to enable asset owners, regulators and policymakers to collaborate using the CReDo digital twin to make decisions which maximise resilience across the infrastructure system rather than from a single sector point of view.

CReDo's purpose is two-fold:

1. To demonstrate the benefits of using connected digital twins to increase resilience and enable climate change adaptation and mitigation
2. To demonstrate how principled information management enables digital twins and datasets to be connected in a scalable way as part of the development of the information management framework (IMF)1

This first phase of CReDo running over the period April 2021 to March 2022 has focused on delivering a minimum viable product to bring the datasets together to offer insight into infrastructure interdependencies and system impact. Separate technical papers have been produced to describe each stage of the project so far:

CReDo Technical Paper 1: Building a cross sector digital twin

CReDo Technical Paper 2: Generating flood data

CReDo Technical Paper 3: Assessing asset failure

CReDo Technical Paper 4: Understanding infrastructure interdependencies and system impact

CReDo Technical Paper 5. Guidance on the use of climate data (UKCP18)

The Technical Reports are nested under the CReDo Overview report, and all Credo reports and related materials can be found on the Digital Twin Hub, https://digitaltwinhub.co.uk/projects/credo.

1.2 Understanding dependencies

This paper describes the work done on the understanding of infrastructure interdependencies and impact on the overall system.

The work on the model described in this report started in September 2021. Access to the data was given at the end of October 2021 and the technical work ran until Mid-January 2022. The work was lead by Lars Schewe and primarily carried out by Mariel Reyes Salazar. The integration of the multiple different networks was carried out by Maksims Abaļenkovs.

We achieved to demonstrate that we can integrate the data from a digital twin into component networks models and could connect these with an overarching coordinating algorithm. This allows us to propagate failures in the networks and then analyze the impacts on the different networks. The observed runtimes for the test networks indicate that the implemented methods will work on realistic networks and that implementing more complex models is feasible in follow-up projects.

2 Overview

The technical work planned in the work package was to model each of the component networks, build models that allow to propagate failures through each of them, and propose methods to propagate the failures between them.

To structure the work, we proposed three levels of detail for the network models and two levels for the integration. In addition, the objective functions for the underlying optimization problems were to be developed.

Due to unavailbility of data and the short timescale, it was decided to focus on the first levels for all networks and the integration. As no data was available that could guide the definition of an objective function, this work was not undertaken.

The basic models were implemented in Python and tested on a small-scale model of part of a UK town. This allowed to demonstrate that the overall methodology is sound and that data from a digital twin can be transferred to more detail network models and the results can be played back to the digital twin.

2.1 Network models

To model the different networks, four levels were proposed:

- basic graph connectivity analysis,
- linear network flow analysis,
- stationary physical models,
- time-dependent physical models.

The first two levels were chosen as they did not require to consider the specifics of the networks, but could already give a basic understanding of the impact and possible recovery actions.

To give an example of the different hierarchy levels, we consider the example of freshwater networks. For the finest level, the models described in [1] form a good starting point. The advantage of these models is that they the same mathematical techniques can be applied to, *e.g.*, sewage networks [2]. The idea is to model the network with a directed graph, so that each pipe and controllable element is an arc in this graph. The flow along each pipe is described by a conservation law. In the case of freshwater networks the commonly used set of equations is known as the waterhammer equations. Other elements, for instance, pumps are described by algebraic equations. The nodes of the network correspond to water sources, demand zones, and reservoirs. To model the connectivity, we make sure that mass is conserved at nodes.

These models lead to mixed-integer optimal control problems, which are, in general, too challenging to solve in practice. Hence, the next simplification step is to disregard the time dependency. In our case, this allows to replace the difficult conservation law with an algebraic equation, which

is known as the Hazen-Williams equation. The mixed-integer nonlinear optimization problems arising from these are still very challenging. Hence, the next simplification step is to linearize the nonlinearity. A very simple version of this is just to limit the amount of flow along each pipe and just retain conservation of mass on the nodes. All of the models so far, require that basic physical parameters of the network are known. For instance, it is necessary to know the diameter of the pipes to obtain a reasonable network model.

The last model disregards all physical information of the model and consists only of the network graph and keeps the node types for information. In this model, the only information that remains is *connectivitiy*. The core query is: Is a demand zone still connected to any water source? This is then a purely qualitative question, no regard is given to the amount of water that the connected sources can provide or the amount of water needed in the demand zone.

In the following, we describe this model in more detail:

The network is given as a directed graph $G = (N, A)$ with nodes N and arcs A. Note that we are allowing parallel arcs. Furthermore, we have a list of supply nodes $S \subseteq N$ and demand nodes $T \subseteq N$. The type of nodes marked as supply or demand depend on the specific network. They might be literal supply nodes, for instance, for fresh water. They might also be connections to an external network, for instance, to the GB electrical grid.

Faults on the network are given by a list of flooded nodes $F_N \subseteq N$ and arcs $F_A \subseteq A$.

The basic fault propagation model removes the nodes F_N and the arcs F_A from G to obtain a graph $\bar{G}$.

The output is the set of all nodes that are either element of F_N or are no longer reachable from supply nodes in $\bar{G}$.

The best algorithm to do this depends on the specifics of the network. In the general case, the Floyd-Warshall algorithm [for a textbook version see, for instance, 3] can be used to compute the transitive closure of the graph, from which the output can be directly read off. The drawback of this algorithm is, however, its worst-case running time of $O(N^3)$.

But typical networks are not general graphs. They are typical sparse, that is, they contain few arcs and only few sources (or few sinks) compared to the total number of nodes. In either of these cases, using depth- or breadth-first-search starting from the sources is a faster option. Algorithms with a faster theoretical running time have been proposed for more restricted graph class, but we did not consider these algorithms here.

In our given data, where supply and demand nodes were not consistently identifiable. Hence, we used a slightly different output definition than the one given above: We are only considering flooded nodes F_N and we output all nodes that are *reachable from a flooded node*. This means that in our case, one pass of a depth- or breadth-first search type algorithm is enough to identify the affected nodes, see Section 3.5.

2.2 Decision support and objectives

The goal of CReDo is, of course, to provide decision support to the asset owners and other stakeholders. For the first phase of CReDo, we chose the one problem as our guiding question: How do we identify those assets that need to be repaired or replaced first in a flood scenario? Due to time and data constraints, we only modelled the basic question and did not implement a full model.

The core situation we chose to model was that the asset owner can only repair a given number of assets and needs to choose which assets to prioritize. We did not consider refinements of the question, for instance, how to deal with assets whose status is unknown, because they have lost communications.

To measure the quality of possible options, we will need to introduce *objective functions*. These will allow us to compare different decisions. In the discussions with asset owners, we identified two promising candidates for further investigation. For both, we did not have enough data to implement them in this iteration of CReDo.

1. Number of households supplied by the network
2. Total "importance" of the supplied nodes in the network.

The first quantity is straightforward to compute given available data. It allows us to prioritize the actions that re-establish service for the most people in the shortest time. It has short-comings: High-priority assets, like hospitals, might be undervalued as their importance cannot be expressed just by the number of directly impacted people. Hence, it was suggested to come up with a more tailored objective that allows for a higher weighting of these high-priority assets. As the data was not available and would be highly sensitive, we did not pursue this further in this iteration of CReDo.

We discuss one possible simple model for this question in Section 4.

2.3 Integrating the networks

As the network models we used were all identical due to data constraints, we could use the integration algorithm for a single network also for the propagating the failures between networks. To keep the structure modular, it was nevertheless implemented as a separate part (see Section 3.5).

3 Implementation

In this section we define the approach taken for modelling the network sites, the technology used in this demonstrator and present the implementation workflow.

While the models discussed in the previous section are easy to write down, the focus of CReDo was that models like this can actually be implemented for real networks. Hence, most of the work in this work package was spent on the implementation. This section discusses the details of the implementation.

The implementation was done in Python 3.8 [4] running in a Podman container [5]. This allowed us to react quickly to changing requirements and deploy the resulting package safely.

Two Python packages were used to provide the core functionality of the software:

- NetworkX [6] for graph representation and graph algorithms, and
- Pandas [7, 8] for providing tabular data storage.

3.1 Data review

The data used by the OR module is obtained from text files that are exported from the Knowledge Graph [9], and further modified by the Expert Elicitation models [10], which update the asset statuses using Bayesian network models. Each subnetwork's assets and connections are stored in text files. The data used are stored in eight different JSON files for each subnetwork of each asset type.

The asset networks, and their subnetworks are the following:

1. Power network
   (a) Primary substations
   (b) Secondary substations
2. Telecommunications network
   (a) Cabinets
   (b) Exchange nodes
   (c) Mobile masts
3. Water network
   (a) Sewage sites
   (b) Sludge site
   (c) Water sites

The sludge site is a single water treatment site in the area of interest and, as such, it is an isolated element in the whole network.

As previously noted, the main information that was accessible to us was the connectivity information: Which assets are connected to each other? Furthermore, it was not consistently possible to extract the property of being a node that is a supply node for, for instance, water or a demand node. Hence, we opted to use a graph theoretic notion: We assumed a node is a supply node, if it only has outgoing arcs and a demand node, if it only has ingoing arcs.

Other information about the physical connection types *etc.*, was not accessible. This prevented us from using more detailed models of the networks that would have allowed us to capture the specifics of the different asset classes.

3.2 Network recreation

The implementation developed for this demonstrator first reads the plain text files from the Knowledge Graph for each asset and recreates the networks following these steps:

1. Read the input data
2. Extract information on assets and their connections
3. Build the network

Reading input data

In this step, the input files are read together and decoded for preprocessing in the next steps of the workflow. The input files are stored in a data structure that distinguishes between the type of site information each file contains. For instance, the primary and secondary substation files are marked as power sites.

This process relies on three environment variables in the DAFNI platform that hold the file names of the asset data.

```
BT_FILES=cabinetmeta.json,exchangemeta.json,mobile_mastmeta.json
UKPN_FILES=primary_substationmeta.json,secondary_substationmeta.json
AW_FILES=sewage_sitemeta.json,water_sitemeta.json
```

Extracting information on assets and their connections

This step uses the input data that were previously decoded, and obtains the following properties from each site in the asset files

- Site information
  - Site ID: The asset identifier used by the asset owners
  - State: A boolean flag which defines whether the site is operational (live, online) or not
  - IRI: The asset identifier used by the Knowledge Graph
  - Subnetwork: The asset subnetwork, *e.g.*, exchange, primary, sewage.

In addition to these properties, the logic implemented initialises the following properties for each site:

- is_sink: A binary flag which defines whether the site is a sink in the network. Initialised with a value of 0.
- is_source: A binary flag which defines whether the site is a source in the network. Initialised with a value of 0.

To find the site connections (network arcs), the following information is gathered from the input files:

- Site connections
  - Site ID: The site identifier used by the asset owners
  - Supplied by: A list of sites which are supplied a flow to this site
  - Supplies to: A list of sites which this site supplies a flow to

Using the “Supplied by" and “Supplies to" connection properties, two lists are generated:

- sink_list: A list of sites that this particular site supplies to. Uses the “Supplies to" property.
- source_list: A list of sites that this particular site receives flow from. Uses the “Supplied by" property.

The following pseudocode presents the logic developed for obtaining the site connections.

Pseudocode: Site connections logic

Input: Site data

Output: sink_list, source_list

```
1  Initialise sink_list = [empty list], source_list = [empty list]
2  for site in sites do
       /* Get the connections, if any                                  */
3      if "Connections" ≠ ∅ then
           /* Find the "Supplied by" sites - sources of this site      */
4          if "Supplied by" ≠ ∅ then
5              for source in “Supplied by" do
6                  source_list.add(source)
           /* Find the "Supplies to" sites - sinks of this site        */
7          if "Supplies to" ≠ ∅ then
8              for sink in “Supplies to" do
9                  sink_list.add(sink)
10     else
           /* This site does not have any connection                   */
11         continue
```

The sink_list and source_list entities are used for building the arcs in the network and used for finding if the site is a sink or a source.

The logic implemented for finding whether a site is a sink or a source is detailed in the following pseudocode. The logic overwrites the initialised values of the is_sink and is_source flags, if

applicable.

Pseudocode: Sink and source logic

Input: sink_list, source_list

Output: is_sink, is_source flags

```
for site in sites do
    /* A sink is a site with incoming connections only */
    if source_list ≠ ∅ AND sink_list = ∅ then
        is_sink = 1
    /* A source is a site with outgoing connections only */
    if sink_list ≠ ∅ AND source_list = ∅ then
        is_source = 1
```

Consolidation of network data

The site information and the site connections are stored in two different pandas dataframes. A *dataframe* is a two-dimensional tabular data structure with labelled axes (rows and columns). The reason for using this data structure is due to its simplicity to handle data in a table-like format, where each row represents a site.

Tables 1 and 2 show a schematic example of the site information and the site connection dataframes, respectively.

Table 1: Schematic of the site information dataframe

Site ID	State	IRI	is_sink	is_source	subnetwork
Site1234	True	http://KnowledgeGraph/Site1234/sewageState	0	0	sewage

Table 2: Schematic of the site connection dataframe

Site ID	sources	sinks
Site1234	[a list of sites]	[a list of sites]

Network arcs

The site connection dataframe is only a precursor as its purpose was only to extract the connections information from the input data. This dataframe will be used for preparing the connections in a format that is useful for the network building stage. The sink_list and source_list of each site are expanded into pairs of (source, sink) sites.

Each pair has the following new properties:

- Arc ID: A numeric identifier for each arc.

- Arc weight: A weight associated with each arc. This weight can be interpreted as a priority. The priority can be defined in terms of users, the sites this arc connects, or if the arc distributes an important flow to a certain building or facility, among other criteria which an asset owner may set.

Finally, the source–sink pairs along with their properties are stored in a new dataframe called `arcs`. This is the dataframe that will be used for the network building. Table 3 offers an schematic example of the arcs dataframe.

Table 3: Schematic of the network arcs dataframe

Arc ID	Source	Sink	Arc weight
123	Site123	Site124	1

Building the network

Prior to building the network objects, all individual subnetworks are concatenated into three large sets of networks—power, telecommunications and water. This approach gives a more adequate vision of the connections between subnetworks (*i.e.* consider a single network of certain asset type). For instance, a primary power station may supply electricity to a secondary substation.

The network built for this implementation is a NetworkX directed graph. The functions developed earlier prepare the graph for receiving the sites and the directed arcs.

The network graph object obtains the site properties associated with each node, *i.e.* Site ID, IRI, is_sink, is_source and state.

The entire network is generated at this stage. It contains all of the sites and arcs from the input files produced by the knowledge graph. This network does not consider the asset statuses, as this network object represents the entire network in its *initial* state (with all assets and their connections present and online).

3.3 Network update

After the network objects are built, it is required to update the networks now taking into account the site statuses, and propagating failures to dependent sites. The statuses are read from the input JSON files. These statuses come from the Expert Elicitation model stage [10].

The network update stage obtains a subgraph of the original networks, by only obtaining the sites that remain in the networks which have a `true` status (*i.e.* the asset is operating) with failure propagation implemented using using a breadth- or depth-first search algorithm as discussed in Section 3.5. The list of lost sites is then computed using standard database operations.

The list of lost sites is exported in the CSV format by outputting only the IRI and its status. This file will be consumed by the knowledge graph and update the networks to show the operating and the non-operating sites accordingly.

This completes the workflow of the connectivity analysis to be added to the minimum-viable-product.

3.4 Single network implementation

We summarize the overall workflow in Figure 1. First the asset files from the knowledge graph are read and, in a second step, the network representations are built (see Section 3.2). Then the flooded assets are identified and removed from the network (see Section 3.3). The final step is to write the lost assets to a file for processing by the knowledge graph.

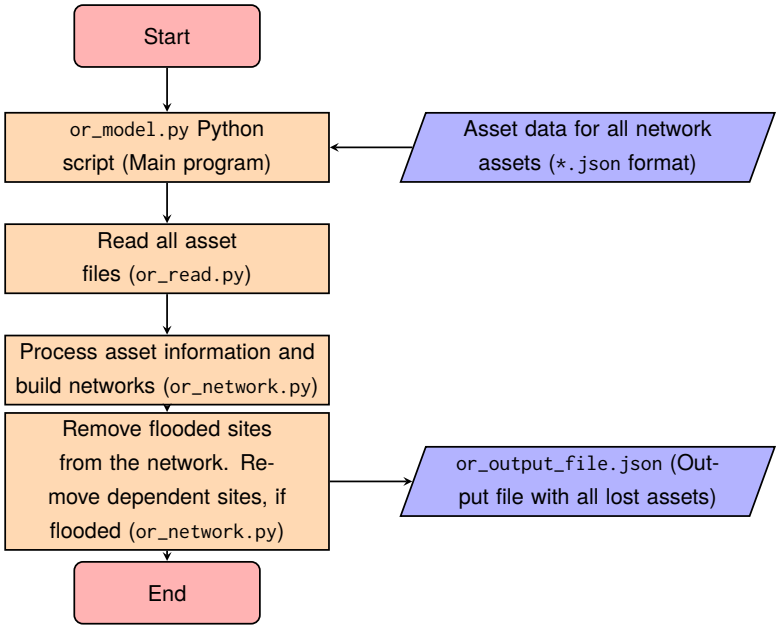


Figure 1: Implementation flowchart of the connectivity analysis for individual networks

3.5 Fault propagation

Fault propagation is a process of gradual expansion of faults over the entire network. A fault in this context is an asset that was affected by severe climate conditions and therefore went offline. The task of fault propagation is to analyse the entire network and take offline all the assets that are connected to faulty assets.

Single type

Fault propagation is implemented as a two phase process. Each phase is encoded in as a separate Python script. The first phase generates dedicated NetworkX graphs for each type of assets, *i.e.* there will be three graphs:

- power,
- phone,

- sewage.

Pseudocode: Graph generator

Data: Json files F, asset types T
Result: Digraphs G
foreach *asset type* $t \in T$ **do**
    initialise digraph g;
    foreach *Json file* $f \in F$ **do**
        add_vertices(g, f);
        add_edges(g, f);
    pbz2_save(g);

This Python script is called `graph-generator.py`. It consists of two functions `add_vertices()` and `add_edges()` called in the main body of the script. Algorithm describes the key steps of the graph generation process. An empty digraph is created for each asset type t, then the algorithm parses all available Json files F looking for assets of a given type t. This search is split into two stages: (a) finding and populating the graph with vertices and (b) finding and populating the graph with edges. Finally, the graph g is saved into a Bzip2-compressed file.

The second phase implements the fault propagation. It is encoded in the Python script called `fault-propagator.py` The algorithm below illustrates all the main steps of fault propagation. At first, a digraph g for a particular asset type t is loaded from a Bzip2-compressed file f. Then the algorithm enters into its first stage: detecting flooded vertices. If vertex u is flooded all of its other states (power, phone and sewage) automatically flip to false. Then vertex u may have neighbour vertices or successors, directly connected to u. The flooding of u will affect all of its successors. Therefore, u is stored in a set F for further processing.

The second stage deals with vertices in the set F. The while loop proceeds until the set F is not empty, *i.e.* it still has vertices to process. The first vertex u is popped from F. Its successors are analysed in the following loop. States of successor v are flipped to false, because v is affected by a flood at u. Consequently, the edge (u, v) connecting u to v is also affected. It should be removed from the digraph. Next, it's time to pay a closer look at v and check whether it is isolated, *i.e.* its in and out degree is 0 (it has no more edges and is not connected to any other vertex in the graph). It is not supplied by anything and does not supply anything to anyone else. Therefore, it should also be removed from the digraph. Otherwise, it still has edges and is still connected to other vertices, it will be added to the end of set F for further processing. Finally, after all successors of u were processed it is time to ask, whether u itself is isolated. If it is—it can be safely removed from the graph.

Multi type

It is possible to create a global graph that would incorporate vertices and edges of all available asset types T. However, we need to track to which underlying networks a given node belongs. In the `NetworkX` world it corresponds to a node v with an attribute "type". This attribute needs to hold more complex data than simply the name of the networks.

Pseudocode: Fault propagator

```
foreach asset type t in T do
    load digraph g from file f;
    /* Detect flooded vertices                                    */
    foreach vertex u ∈ g do
        if u is flooded then
            set ∀ states(u) := False;
            save F ← u;
    /* Process flooded vertices                                   */
    while F is not empty do
        pop u ← F;
        foreach successor v of g(u) do
            states(v) := False;
            remove edge (u, v) from g;
            if v is isolated then
                remove vertex v from g;
            else
                save F ← v;
        if u is isolated then
            remove vertex u from g;
```

Our approach was inspired by the Unix file permissions [11]. In Unix operating system each file has the following access modes: no access, read, write and execute. These modes are usually abbreviated to `-`, `r`, `w` and `x`. Alternatively, each mode is also encoded and represented by an octal number: no access as `0o0`, read as `0o4`, write as `0o2` and execute as `0o1`[1]. Table 4 summarises this notation. A feature of such an encoding is that any of eight possible mode combinations can be represented by a unique octal number. This unique combination is obtained by an addition or subtraction of mode numbers. In the octal world these "addition" and "subtraction" operations are represented by bitwise `or` and `xor` operators. In Python syntax these bitwise `or` and `xor` operators are "`|`" and "`^`".

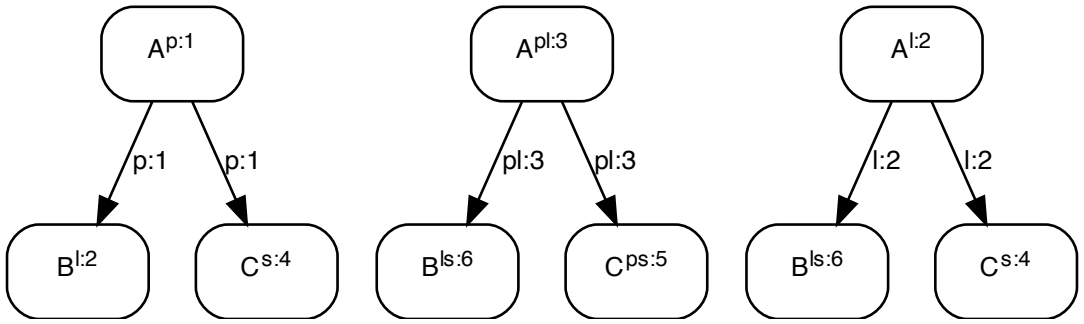
Code	Mode	Octal
-	no access	0
r	read	4
w	write	2
x	execute	1

Table 4: Unix file permissions

Encoding Credo asset types with octal numbers enables multi-type support in `NetworkX`. Analogously to Unix file permissions, the octal codes are 1 for power, 2 for landline phone and 3 for sewage assets. The multi-type version of fault propagator generates a single graph. This global graph consists of vertices and edges representing assets (and their connections) from all Json

[1] Octal numbers are written in Python notation.

files available. Both vertices and edges in this graph have an additional parameter called “Code”. This parameter stores the octal number encoding the type or multiple types of assets.



(a) Scenario 1 *(b) Scenario 2* *(c) Scenario 3*

Figure 2: Multi-type assets

Figure 2 clarifies, how octal codes are used to encode multi-type assets. Consider Figure 2a. It shows a miniature graph that consists of three vertices A, B and C. Vertex A is of type power and has the code 1, vertex B is of type landline phone and has the code 2, while vertex C is of type sewage and has the code 4. There are two “power” connections coming from vertex A to B and from vertex A to C. Therefore, both edges (A, B) and (A, C) are of type “power” and have the code 1.

Refer to Figure 2b. In this case vertex A became a source of a landline phone connection in addition to its existing power source. Therefore, its code changed to `0o3 = 0o1 | 0o2`. Similarly, the edges obtained the same modification and have the code `0o3` now. Vertex B became a sink of a landline connection in addition to its existing sewage state. That’s why its code is `0o6 = 0o4 | 0o2`. Finally (reading new information from Json files) it was discovered that vertex C is also a secondary source of power in addition to its sewage state. Therefore, its code has to update to `0o5 = 0o4 | 0o1`.

Last example in the series depicted in Figure 2c illustrates an opposing scenario, the case when assets become flooded and go offline. Vertex A loses its power source and its code is updated to `0o2 = 0o3 ^ 0o1`. Similarly, the edge (A, B) and (A, C) codes are also downgraded to `0o2`. Vertex B is left intact. Finally, vertex C also loses its power and its code changes to `0o4 = 0o5 ^ 0o1`.

Fault propagation over a global graph with multi type assets requires additional changes to the algorithms explained above. However, the core logic of the algorithms shown in pseudocode remains the same.

It is important to note that this multi type asset support may be extended further. In case more asset types are added to the system, more bits should be added to the code. Current three-type system can be encoded by an octal, three-bit number 2^3, a four-type system will be expressed by a four-bit number 2^4 *etc.*

4 Decision support

For decision support models, we focussed on a slightly more general network model than the model outlined in Section 2. We do not only consider connectivity, but also use a linear flow model with arc capacities.

Given a directed graph $G = (N, A)$ with capacities $c \in \mathbb{R}^A$, such that only a certain amount of flow is allowed to travel in the arc A. These capacities should be derived from the physical capacities of the lines, where applicable. In the case of a fresh water network, this would be a restriction on the volumetric flow rate. We also define a variable f for the flow over each arc, *i.e.* $f \in \mathbb{R}^A \geq 0$. For every flooded arc $a \in F_A$, we introduce a variable $x_a \in \{0, 1\}$ to model whether the arc should be prioritized for repair.

To model supply and demand, we need to consider three types of nodes:

- For all supply nodes $n \in S$, we want the outflow out of the node to be between 0 and an upper bound s on the supply capcacity.
- For all demand nodes $n \in D$, we want the inflow into the node not to exceed an upper bound on the demand capcacity d, but otherwise be as large as possible.
- For all other nodes, inflow should equal outflow.

To model these requirements, we introduce a variable $b_n \in \mathbb{R}$ for every node $n \in N$, which we call the excess of that node. This leads us to the following constraints on the flow:

Flow balance Inflow equals outflow plus excess.

Arc capacity The flow in the arc cannot exceed the arc capacity.

Arc repair Flooded arcs can only be used if they have been repaired.

Supply/demand bounds For supply nodes, the excess is between $-s$ and 0, for demand nodes between 0 and d, and for all other nodes it equals 0.

As discussed, we assume that we can measure the utility of satisfied demand. We formalize this assumption by assuming that for every demand node n, we are given a function $u_n(b)$, that maps a given excess b to a utility. Depending on the available data, we would propose to choose u_n to be linear or at most piecewise linear concave.

A basic mixed-integer optimization model would then look as follows:

$$
\begin{aligned}
\max \quad & \sum_{n \in D} u_n(b_n) \\
\text{such that} \quad & \sum_{a \in \delta^{\text{in}}(n)} f_a = b_n + \sum_{a \in \delta^{\text{out}}(n)} f_a \\
& 0 \leq f_a \leq c_a \quad \text{for all } a \in A \\
& 0 \leq f_a \leq x_a c_a \quad \text{for all } a \in F_A \\
& 0 \leq b_n \leq d_n \quad \text{for all } n \in D \\
& -s_n \leq b_n \leq 0 \quad \text{for all } n \in S \\
& b_n = 0 \quad \text{for all } n \in N \setminus (S \cup D) \\
& f \geq 0, x \in \{0,1\}^{F_A}.
\end{aligned}
$$

For the above choices of u_n, this leads to a mixed-integer linear optimization problem, which can be modelled and solved, at least for small to medium sized networks, using standard solvers. Given more data and time, we would have implemented this model using the modelling framework Pyomo [12] and used the solver CBC to solve the resulting optimization problem.

5 Recommendations

In this demonstrator project we were able to analyse the network connectivity for three major networks—power, water and telecommunications—after a flood occurs in a city. The connectivity analysis was done using a single network approach and combined with a global fault propagation. We hope this technical study shows how network models can be developed for a better understanding of the impact of climate change on our infrastructure.

Several lessons were learned while developing this study which can supplement future iterations of the digital twin project:

1. It is necessary to integrate the data requirements of more detailed models early on in the design process as it will be difficult and costly to request additional data later. We were only able to achieve a fraction of the original goals with the limited amount of data that was requested in the original data request. Additional data would facilitate further work in this area.
2. The achievable detail level of the models is highly tied to the desired spatial and temporal resolution. This needs to taken into account when planning data acquistion and planning for modelling work. In our case, the final area that was covered by the demonstrator was too small to show interesting network interactions.
3. Having easy access to the asset owner data in a tabular format would have reduced the work in this package. The existence of an abstraction layer between asset owner data and our model proved to be an obstacle in our project. In our case, a network for a single asset has the advantage that there exist only a very small number of node and arcs types. Having access to one table per type will reduce implementation time. In the current version, first the asset owner data is processed into a unified representation, in our case, the knowledge graph. We then have to extract the asset-specific data (for instance, the node type) from this unified representation. This means that we have to undo a lot of the work done in the step before, which complicates the workflow.
4. Our modelling assumptions for fault propagation need to refined. We are taking a very simplistic approach to failure: an asset fails if one asset fails it depends on. This does not consider that certain assets can replace each other or that certain failures are acceptable for a short amount of time. Propagation of faults needs to consider different timescales and multiple types of faults. This might be achieved by using multiple networks coupling the assets, for instance for different timescales and fault types. Whether this is a feasible approach needs to be studied in future projects.
5. The current multi-type fault propagation algorithm assumes that all networks can be integrated into one big network. This assumption does not allow for more detailed models of the separate asset-owner networks and makes it necessary to share more information between networks than strictly necessary. Hence, more thought needs to be put into making sure that fault-propagation uses only the minimum amount of information necessary.

6. The network connectivity analysis could be further expanded to include other asset types such as transportation links, *i.e.* road or train networks. These links will decisively impact on the recovery speed of the sites, as human resources will be required to repair a site and bring it online. Whether existing data sets are suitable for this task or it is better to identify bottlenecks through expert elicitation needs to investigated further.
7. The input JSON files could be further expanded to include a network identifier, *i.e.* water, power, telecommunications, to automate the file reading and preprocessing functions. Currently, the input file names are stored in environment variables to aid in this process; yet, an internal field within the JSON files could eliminate the need of the environment variables.

In future studies we aim to expand on the lessons learned detailed above and develop more sophisticated models to show how a real infrastructure network would react to climate change.

Future iterations would benefit from using the model outlined in Section 4. While it is essential to know how failures in different systems interact, it will be even more critical to know where the focus for fixing these failures need to be put. This also needs to incorporate careful modelling of the impact of failure of assets to make sure that societal inequalities are not amplified during disasters.

Nomenclature

CBC COIN-OR Branch-and-Cut, an open-source mixed integer programming solver

CReDo Climate Resilience Demonstrator

CSV Comma Separated Values

DAFNI Data & Analytics Facility for National Infrastructure

IRI Internationalised Resource Identifier

JSON JavaScript Object Notation. An open standard file format

OR Operational Research

References

[1] A. Martin *et al.*, Eds., *Mathematical Optimization of Water Networks*, 1st ed., ser. International Series of Numerical Mathematics. Springer-Verlag, 2012, vol. 162, ISBN: 9783034804356.

[2] F. Hante, G. Leugering, A. Martin, L. Schewe and M. Schmidt, "Challenges in optimal control problems for gas and fluid flow in networks of pipes and canals: From modeling to industrial applications," in *Industrial Mathematics and Complex Systems*, ser. Industrial and Applied Mathematics, P. Manchanda, R. Lozi and A. H. Siddiqi, Eds., Springer, 2017, pp. 77–122. DOI: 10.1007/978-981-10-3758-0_5.

[3] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. USA: Prentice-Hall, Inc., 1993, ISBN: 013617549X.

[4] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009, ISBN: 1441412697.

[5] M. Heon *et al.*, *Podman - : A tool for managing OCI containers and pods*, version v1.0 and beyond. Currently at v3.0.1, Jan. 2018. DOI: 10.5281/zenodo.4735634.

[6] A. A. Hagberg, D. A. Schult and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11–15.

[7] W. McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[8] The pandas development team, *Pandas-dev/pandas: Pandas*, version 1.1.4, Feb. 2020. DOI: 10.5281/zenodo.3509134.

[9] J. Akroyd *et al.*, *CReDo Technical Paper 1: Building a cross sector digital twin*, Centre for Digital Built Britain (CDBB), 2022.

[10] C. J. Dent, B. Mawdsley, J. Q. Smith and K. Wilson, *CReDo Technical Paper 3: Assessing asset vulnerability*, Centre for Digital Built Britain (CDBB), 2022.

[11] A. S. Tanenbaum and H. Bos, "Modern operating systems," in 4th ed. Pearson Education, 2015, ch. 9.3 Controlling Access to Resources, pp. 602–611.

[12] W. E. Hart *et al.*, *Pyomo–optimization modeling in Python*, 2nd ed. Springer Science & Business Media, 2017, vol. 67.

Version Control

Version	Date	Author	Status	Change Description
0.91	February 15, 2022	Lars Schewe		Incorporating feedback
0.9	February 14, 2022	Lars Schewe		Streamlining the text
0.8	February 11, 2022	Maksims Abaļenkovs	θ	Multi type fault propagation was described
0.7	February 9, 2022	Lars Schewe		Finalized all sections, overall streamlining
0.6	February 4, 2022	Lars Schewe	ζ	Introduction and overview added, conclusion updated, still WIP
0.5	February 1, 2022	Maksims Abaļenkovs	ϵ	Section on overall implementation was added
0.4	February 1, 2022	Maksims Abaļenkovs	δ	Section on fault propagation was completed
0.3	January 31, 2022	Mariel Reyes Salazar	γ	Added environment variables and minor corrections
0.2	January 31, 2022	Maksims Abaļenkovs	β	Text proof-read and corrected
0.1	January 18, 2022	Mariel Reyes Salazar	α	Document created

Authors and Contributors

Lead Author

Lars Schewe

Contributors

Mariel Reyes Salazar
Maksims Abaļenkovs
Chris Dent

The Centre for Digital Built Britain at the University of Cambridge's National Digital Twin Programme is funded by the Department for Business, Energy and Industrial Strategy via UK Research and Innovation.