DEPARTMENT OF MECHANICAL ENGINEERING

MECH0020 Individual Project

2019/20

Student:	**Bowen Fan (17038667)**
Project Title:	**Analysis and Optimisation of Water Distribution Systems using EPANET**
Supervisor:	**Dr Yuriy Semenov**

Word count: 7477

University College London
Torrington Place
LONDON WC1E 7JE

Analysis and Optimisation of Water Distribution Systems using EPANET

UCL Mechanical Engineering Third Year Project

Bowen Fan

Supervisor: Dr Yuriy Semenov

13 May 2020

Abstract

In this project, two algorithms for optimising water distribution systems (WDS) were developed. Both algorithms make calls to EPANET, a hydraulic modelling software written by the US Environmental Protection Agency, to evaluate the hydraulic performance of a WDS. They are implemented as Python programs, whose source codes and compiled executables are given in the Appendix.

The first algorithm solves the pipeline diameter optimisation problem: Given a set of commercially available pipe diameters to choose from, there exists an optimal least-cost configuration of diameters that can fulfil pressure and flow requirements. For real-world water distribution networks, this problem is impossible to solve with a brute-force exhaustive search as the search space scales exponentially.

Thus, a genetic algorithm has been developed to find near-optimal configurations within reasonable computing time. The genetic algorithm achieves this by simulating the process of natural selection: configurations are selected by cost and feasibility, and the best networks are merged to produce children configurations which possess advantages from both parents. Thus, the results improve over a large number of generations and a near-optimal configuration is discovered.

The second algorithm is novel. It solves the buffer tank location optimisation problem: As water demand varies through the day, demand-balancing tanks (also called buffer tanks) are required in the network to smooth out demand variations. They are filled while demand is low and release their contents during demand peaks. However, to achieve the best buffering effect, they must be placed at optimal locations in the network. Thus, an algorithm has been developed to exhaustively search the entire network to find the best node location to add a buffer tank.

The tools developed in this project will be made open-source and available for WDS community engineers, by publishing this report on arXiv and releasing the programs as an extension block of the EPANET software. It is hoped that, beyond this project's academic findings, it can be a tiny contribution to the benefit of WDS engineers everywhere.

Declaration

I, Bowen Fan, confirm that the work presented in this report is my own. Where information has been derived from other sources, I confirm that this has been indicated in the report.

Acknowledgements

I wish to thank my supervisor, Dr Yuriy Semenov, for the opportunity to do this project. This is the single biggest project I have done as a student, and is also the one where I have learnt the most. Through this project, I have learnt a great deal in water distribution systems, hydraulic modelling, computational optimisation methods and programming. I am really happy to have chosen this project.

Beyond just providing the opportunity, Dr Semenov has also guided me throughout the project, while still allowing me the freedom to choose the specific areas in which I wish to focus. I wish to thank him for his encouragement and advice, both for this project and for my career and academic pursuits.

Contents

Nomenclature

Abbreviations

BIN Balerma Irrigation Network: model of the irrigation WDS in Balerma, Spain

CFS Flow or demand in cubic feet per second

DDA EPANET's demand-driven analysis engine

GA Genetic algorithm

GGA Global gradient algorithm

HAN Hanoi Network: simplified model of the WDS in Hanoi, Vietnam

LPS Flow or demand in litres per second

MOD Modena Network: model of the WDS in Modena, Italy

PDA Pressure-dependent analysis

TLN Two-Loop Network: An example theoretical WDS

WDS Water distribution system

Sets

$\mathbb{D}$ Set of commercially available discrete pipe diameters

$\mathbb{N}$ Set of nodes in the WDS

$\mathbb{P}$ Set of pipes in the WDS

$\mathbb{S}$ Set of possible solutions, i.e. search space, of an optimisation problem

Symbols

$\bar{P}_{i,node}$ Time-averaged pressure of a node i in m

D_i Diameter of a pipe i in mm

E_i Elevation of a tank or a node i in m

L_i Length of a pipe i in m

N_d Number of commercially available discrete pipe diameters

N_p Number of pipes in the WDS

List of Figures

List of Tables

1. Introduction

1.1. Introduction to Water Distribution Systems

A water distribution system (WDS) is a critical part of any city's infrastructure. Its role is to deliver potable water from water treatment plants and service reservoirs to household and commercial consumers. All water distribution systems comprise a network of pipelines, pumps, and facilities for water storage (e.g. reservoirs and tanks). Apart from routine water consumption, these systems must be capable for supplying adequate water for firefighting purposes.

A WDS must in general fulfil three types of criteria: it has to maintain physical, hydraulic, and water quality integrity. Physical integrity refers to preserving a physical barrier to keep contaminants out. Hydraulic integrity pertains to meeting pressure and velocity requirements throughout the network, so that consumers have a reliable supply of water. Water quality integrity is concerned with keeping contaminants (e.g. sedimentation, pesticides, coliform bacteria) within safe levels throughout the WDS.

This thesis will focus on the analysis and optimisation of a network's hydraulic performance. Nevertheless, these three criteria are inter-dependent. For example, excessive hydraulic heads in the network increases the risk of pipe and valve failure, thus compromising the physical integrity of the network. When the physical integrity is compromised, contaminants are likely to enter, thereby impacting water quality.

The scale of a WDS in a large city like London is impressive: Thames Water, the water utility for Greater London, supplies to the city an average of 2.6 billion litres of drinking water per day. This quantity is channelled to the users via 31,000 km of transmission pipeline (Thames Water, 2017).

With a network of this scale, even a small optimisation of its cost and reliability will make a big difference. However, it is precisely the scale and complexity of the network that makes optimising it difficult. Hence, this project is dedicated to the analysis and optimisation of the cost and hydraulic performance of water distribution networks. It makes extensive use of EPANET, a hydraulic modelling software.

1.2. Introduction to EPANET

EPANET is a public domain hydraulic modelling software written by the United States Environmental Protection Agency. It is able to simulate network performance in terms of hydraulics and water quality, by calculating network characteristics such as pressure, flow, and water age.

The WDS is defined in EPANET by a system of nodes and links. Nodes represent storage facilities and demand junctions, while links represent pipes, valves, and pumps.

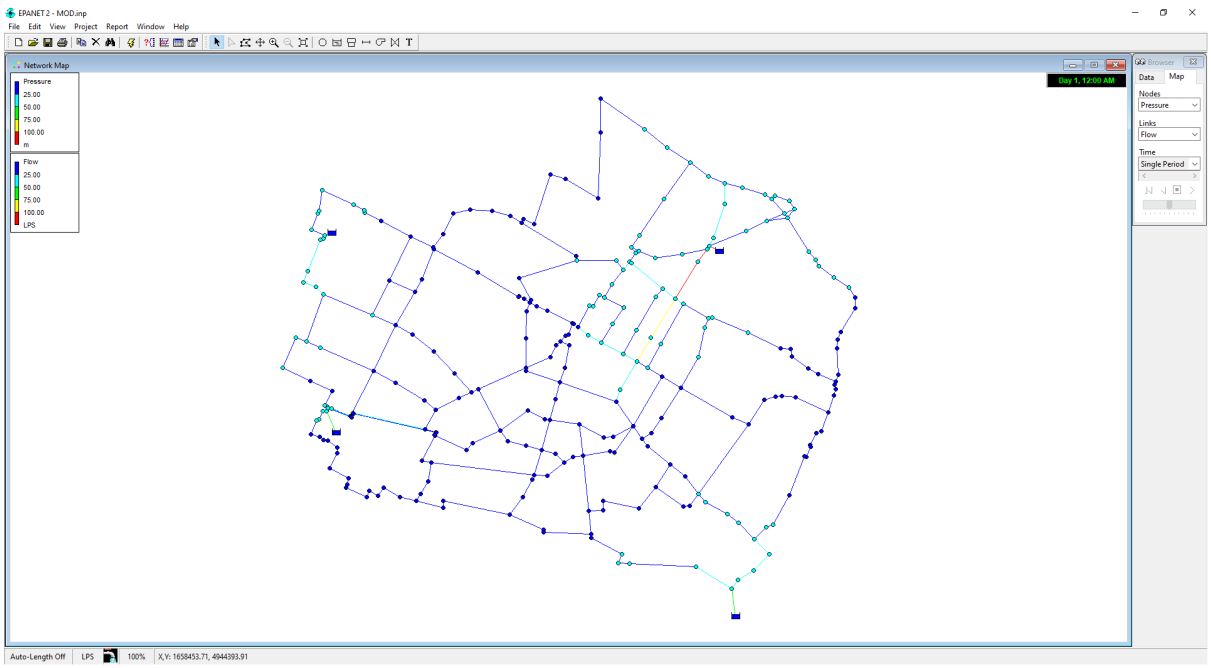


Figure 1.1.: EPANET Program Interface

1.3. Introduction to WDS Optimisation

Water distribution systems are intricate systems which are expensive to construct and maintain. Thus, there is need to optimise their design so that they can meet pressure and flow requirements with minimal cost. Multiple elements of the system can be optimised: the configuration of pipe sizes, pump scheduling, placement of buffer tanks, pipe material selection, etc.

This project is concerned with optimising pipe sizes and buffer tank locations. The optimisation of pipe diameters has been studied in literature by multiple authors. However, the authors do not publish their source code, and rarely package their work into usable programs. Most programs that were released are now obsolete and cannot be

executed on a modern platform. There is therefore a need to develop a new program whose algorithm performance can come close to what have been achieved in literature.

Buffer (or demand-balancing) tanks are needed in any real-world network to smooth out demand imbalances. They are filled when demands are low and network pressures are high, then release the stored water to supplement the main reservoirs during peak demand. To achieve the best balancing effect, they need to be placed at optimal locations in the network. There is a surprising lack of research on this problem. Thus, there is a need for an efficient algorithm to determine optimal locations for adding demand-balancing tanks.

1.4. Aims and Objectives

The aim of this project is to analyse, then optimise, real-world water distribution networks with the help of EPANET.

1. First, the characteristics of a real-world network will be studied using EPANET. A literature review of network design considerations and network optimisation methods will be conducted.
2. An optimisation method to find the least-cost configuration of pipe diameters, subject to pressure and velocity requirements, will be developed. This will be implemented as a Python program which utilises the EPANET Programmer's Toolkit for hydraulic analysis.
3. To improve the network's ability to cope with varying demands, an optimisation method to determine the best location for a demand-balancing tank will be developed. This will also be implemented as a Python program which calls the EPANET library for hydraulic analysis.
4. Finally, the limitations and inaccuracies of EPANET encountered in this project will be identified and studied. Potential improvements will then be proposed.

1.5. Thesis Organisation

This thesis comprises five main sections, including this introduction chapter. The sections and their contents are:

Chapter 2 elaborates on the theory and engineering equations behind hydraulic modelling methods, and presents a literature review of WDS analysis and optimisation

research.

Chapter 3 is on the pipe diameter optimisation problem and covers the methodology used to develop the solving algorithm. It then presents the results of the algorithm, and a comparison of these results to literature.

Chapter 4 is on the buffer tank location optimisation problem. It covers the methodology used to develop a novel algorithm for finding the best location for a buffer tank. The results of the algorithm are evaluated by running it on a complex real-world network and studying its efficacy.

Chapter 5 is the concluding chapter. It includes a brief discussion of the results and the project's limitations. Finally, potential future extensions to this project will be discussed.

The source code of the programs will be presented in the Appendix, along with links to their hosting pages on GitHub.

2. Theory and Literature Review

2.1. Hydraulic Modelling Methodology

EPANET uses the "Global Gradient Algorithm" (GGA), adapted from the Newton-Raphson method by Todini and Pilati (1987), for solving network hydraulics in discrete timesteps. As described by the author of EPANET, Rossman (2000), the "hybrid node-loop approach" is used to set up a system of conservation equations, which the GCA is then called to solve. In essence, the principles behind the node-loop algorithm is analogous to Kirchhoff's current and voltage conservation laws within a circuit (Belova et al., 2014).

To elaborate, EPANET sets up a system of non-linear mass and energy conservation equations for each loop within the network (Rossman, 2000). The energy balance equations make use of either the Hazen-Williams or Darcy-Weisbach equation to calculate the head loss between two nodes. EPANET converts all hydraulic heads to ft and flow rates to CFS so that the following equations, designed with ft as the base unit, can be employed.

Hazen-Williams head loss equation used in EPANET (Rossman, 2000):

$$\Delta h = \frac{4.727\ L\ q^{1.852}}{C^{1.852}\ d^{4.871}} \tag{2.1}$$

Where Δh is the head loss (ft), L is the pipe length (ft), q is the flow rate (CFS), d is the pipe diameter (ft), and C is the Hazen-Williams coefficient (around 130 for cast iron pipes, 140 for PVC pipes).

Minor loss equation (Rossman, 2000):

$$\Delta h = K\frac{v^2}{2g} \tag{2.2}$$

Where K is the minor loss coefficient (empirically determined for various network elements and pipe shapes), and v is the flow velocity (ft/s).

The total head loss between two nodes is given by the sum of major and minor losses:

$$\Delta h = \frac{4.727\ L\ q^{1.852}}{C^{1.852}\ d^{4.871}} + K\frac{v^2}{2g} \tag{2.3}$$

Having set up a system of energy-balance equations, EPANET now establishes the system of mass-balance equations, based on the flow continuity principle: water entering the node = water exiting – nodal demand withdrawn (Rossman, 2000):

$$Q_i - Q_j = D_i \quad \forall\, i, j \in \mathbb{N} \tag{2.4}$$

Where Q_i is the flow into node i, Q_j is the flow from i to j, and D_i is the demand withdrawn at node i. $\mathbb{N}$ is the set of all nodes in the network.

EPANET then sets up a matrix equation that combines the head loss equations in a pipe loop with the continuity equations around all nodes. This is a non-linear system of equations that has to be solved iteratively. EPANET then uses the Global Gradient Algorithm to solve for convergence. The solution that it derives is valid on the assumption that the system is at a steady-state.

For analyses over a longer period of time, e.g. several hours, EPANET breaks down the period into multiple steady-states. The system jumps between the steady-states in a step-wise manner. As such, EPANET is unable to model transient events like pipe bursts and water hammering.

2.2. EPANET Programmer's Toolkit

EPANET's computation engine is made available to external applications through a library of functions, known as the Programmer's Toolkit. This is an Application Programming Interface (API) that provides all of EPANET's functions to an external program.

For example, a Python program can call the API to solve network hydraulics, then retrieve network properties such as pressure and flow for all nodes and pipes. The pipe diameter and buffer tank optimisation algorithms developed in this project relies on calling the API to solve hydraulics and retrieve properties.

2.3. Literature Review on EPANET's Limitations

The most crucial limitation of EPANET is, arguably, its inability to correctly model pressure deficient conditions in the network (Sayyed et al., 2014). To solve the system

of mass and energy balance equations, EPANET first assumes that the nodal demands are always met, then uses those demand values to calculate the corresponding nodal pressure that will balance the energy conservation (Gupta and Bhave, 1996). This is known as a Demand-Driven Analysis (DDA).

In reality, it is pressure that drives flow. Hence, DDA will return incorrect results when there is insufficient pressure to satisfy a node's flow requirement: the nodal pressure will be calculated as negative, meaning that the *demand* node is modelled to be *supplying* water into the network (Ang and Jowitt, 2006).

This limitation has been studied extensively, and various solutions have been developed. Kalungi and Tanyimboh (2003) developed a Pressure-Dependent Analysis (PDA) algorithm which models the actual flow received to be parabolically dependent on head surplus. Babu and Mohan (2012) developed a workaround method that adds artificial reservoirs and flow control valves to pressure-deficient nodes, allowing their flow to be more accurately modelled. Later works (Gorev and Kodzhespirova, 2013), (Sivakumar and Prasad, 2014) made further improvements and applied the PDA algorithm to study real-world networks.

2.4. Literature Review on WDS Optimisation

There exists considerable academic research on WDS optimisation, though authors rarely implement their research results as executable programs. Multiple algorithms have been used to tackle the pipe selection problem, ranging from earlier methods such as linear and dynamic programming, to newer methods such as genetic algorithms and ant colony optimisation.

Savic and Walters (1997) were one of the earliest researchers to apply genetic algorithms to pipeline optimisation. Their results proved superior to the linear programming algorithm used by Alperovits and Shamir (1977): for a simple two-loop network (illustrated and examined in Chapter 3), the least-cost feasible network found by linear programming costs $497,525, while the genetic algorithm discovered a $420,000 solution. Maier et al. (2003) then applied an Ant Colony Optimisation algorithm, which were able to discover closer solutions to the global optimum for large networks.

Compared to the extensive research on pipeline optimisation, research on buffer tank location optimisation is lacking. Mays (1999) states that, as a 'rule-of-thumb', large buffer tanks should be added to central nodes with the greatest water demands. Seyoum and Tanyimboh (2014) used an evolutionary algorithm to optimise tank "design,

rehabilitation, and operation" to "improve water quality". However, the optimisation of location to improve hydraulic performance was not explored.

Basile et al. (2009) proposed a methodology for tank sizing and operation to improve a small WDS. Again however, no algorithm for location optimisation was presented. The two nodes where the authors studied the addition of a tank were simply "chosen for their proximity to the fire hydrant and their central position on the network", echoing the rule-of-thumb emphasising demand and centrality. In Chapter 4, this thesis shall demonstrate that this rule-of-thumb may not yield optimal results.

3. Optimisation of Pipe Diameters

3.1. Problem Definition

The pipeline diameter optimisation problem is easy to understand but hard to solve: Given a WDS with number of pipes N_p and a number of commercially available diameters N_d, find the cheapest network configuration that can meet pressure and flow requirements.

Smaller pipes are cheaper but larger pipes can supply greater pressure. Thus, an optimal balance must be found.

Formally, the problem is:

$$\underset{D \in \mathbb{D}}{\text{minimize}} \quad C(L, D) = \sum_{i=1}^{N_p} L_i \, c(D_i) \tag{3.1a}$$

$$\text{subject to} \quad P_n \geq P_{min} \qquad \forall \, n \in \mathbb{N}, \tag{3.1b}$$

$$V_{min} \leq V_p \leq V_{max} \qquad \forall \, p \in \mathbb{P} \tag{3.1c}$$

$C(L, D)$ is the network cost, as a function of its constituent pipe lengths and diameters. L_i is the pipe's length, while $c(D_i)$ is a subfunction that retrieves the cost per unit length from a schedule listing the commercial pipe diameters and their prices.

P_n is the pressure at each demand node, V_p is the flow velocity through each pipe. P_{min} is the minimum pressure requirement, while V_{min} and V_{max} are the minimum and maximum velocities allowed in each pipe.

$\mathbb{D}, \mathbb{N}, \mathbb{P}$ are the sets of discrete diameters, nodes, and pipes respectively.

This problem is hard to solve for real networks because the set of possible solutions, i.e. its search space, increases exponentially with network size. In general, the search space $\mathbb{S}$ of the problem has size:

$$|\mathbb{S}| = N_d^{N_p} \tag{3.2}$$

3.2. Methodology: Genetic Algorithms

Genetic algorithms (GAs) are search heuristics for finding a near-optimal solution within a large search space. They simulate natural selection in biological evolution, and are part of a larger group of biologically-inspired optimisation algorithms, known as evolutionary algorithms.

A genetic algorithm is an iterative process, wherein the following steps are repeated to evolve towards the best solution, until a termination criterion is met:

1. Initialisation of the population
2. Selection of the best individuals in the population
3. Crossover between the best individuals
4. Mutation of the results (to avoid being trapped in local minima)

The algorithm is terminated when the number of iterations has reached a predefined limit, and/or the results stagnate over several generations.

The section below illustrates how a genetic algorithm can be applied to solve the pipeline optimisation problem. For the purpose of illustration, a simple network, called the Two-Loop Network (TLN), is used. It consists of 7 nodes and 8 pipes, and there are 14 pipe diameters to choose from for each pipe.

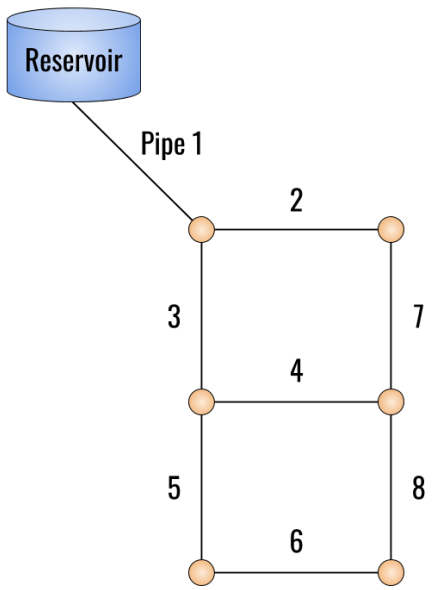


Figure 3.1.: Two-Loop Network

For the Two-Loop Network:

$$|\mathbb{S}| = 14^8 \tag{3.3}$$

The search space of this network is relatively small, and a simple brute-force search will be capable of finding the optimal diameter configuration in a reasonable time. However, even for a modest real-world network consisting only of a few hundred pipes (e.g $N_p = 300$, $N_d = 10$), $|\mathbb{S}| = 10^{300}$: A brute-force search quickly becomes impossible.

The subsections in the following pages explain the methodology of genetic algorithm optimisation.

3.2.1. Initialisation of the population

First, a population of the same base network, but with different diameter configurations, is initialised. For each individual network in the population, the pipe diameters are sampled randomly from the list of available commercial diameters.

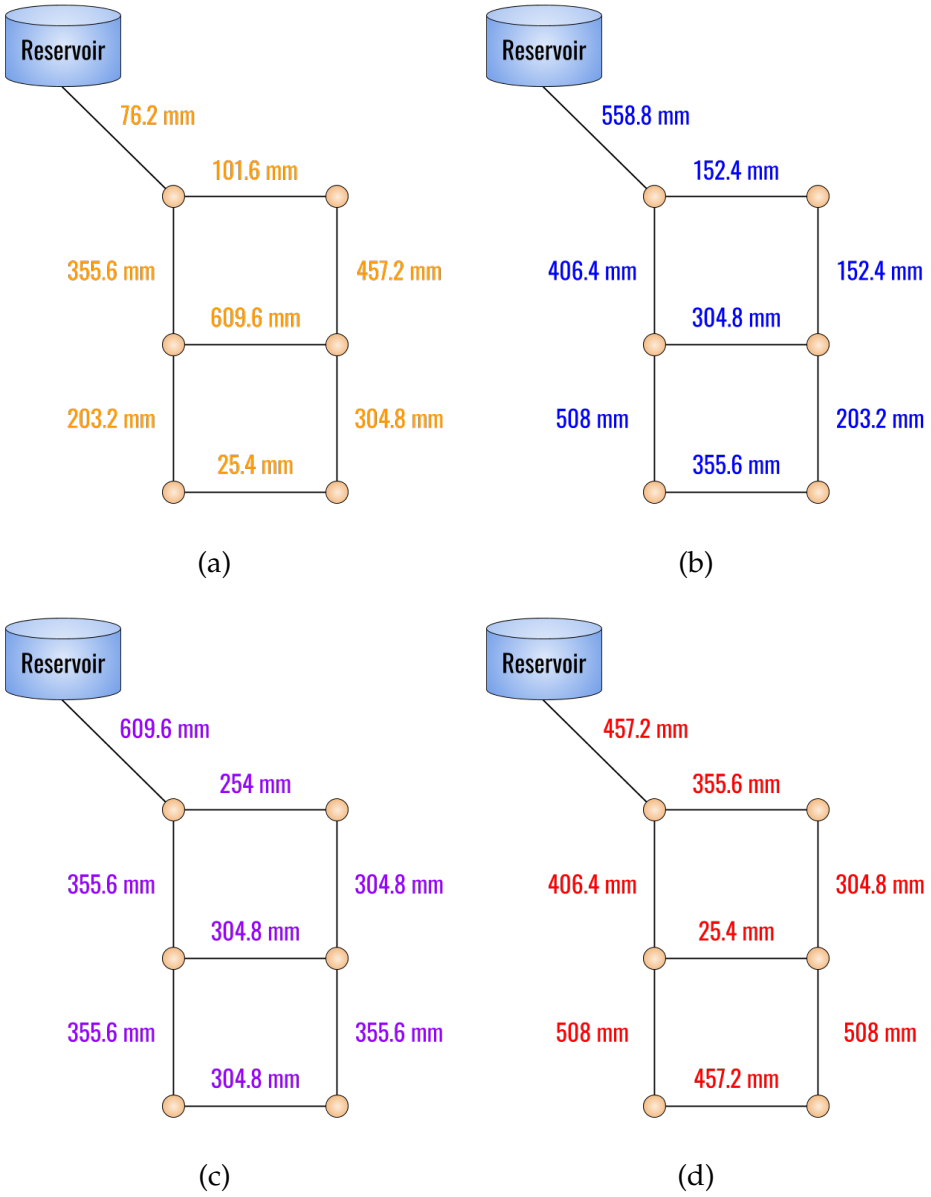


Figure 3.2.: Initialisation

3.2.2. Selection

In the selection step of the genetic algorithm, two evaluations are performed for each individual in the population. First, the cost of each configuration is evaluated by the cost function in Equation 3.1a.

Next, the algorithm calls the EPANET library to evaluate whether the constraints defined by Equations 3.1b and 3.1c are met for each network. The EPANET library runs a hydraulic analysis on each network, then a subroutine in the algorithm retrieves the pressure and velocity values for all nodes and pipes. This subroutine returns a Boolean value: True if the network passes the constraints, and False if it fails.

From the networks that pass, the least-cost individuals are selected, so that they can be combined into children networks which are expected evolve towards a more optimal solution. In this example, 2 of the 4 networks are selected. However, in the actual implementation, the 4 least-cost networks within the population are selected.

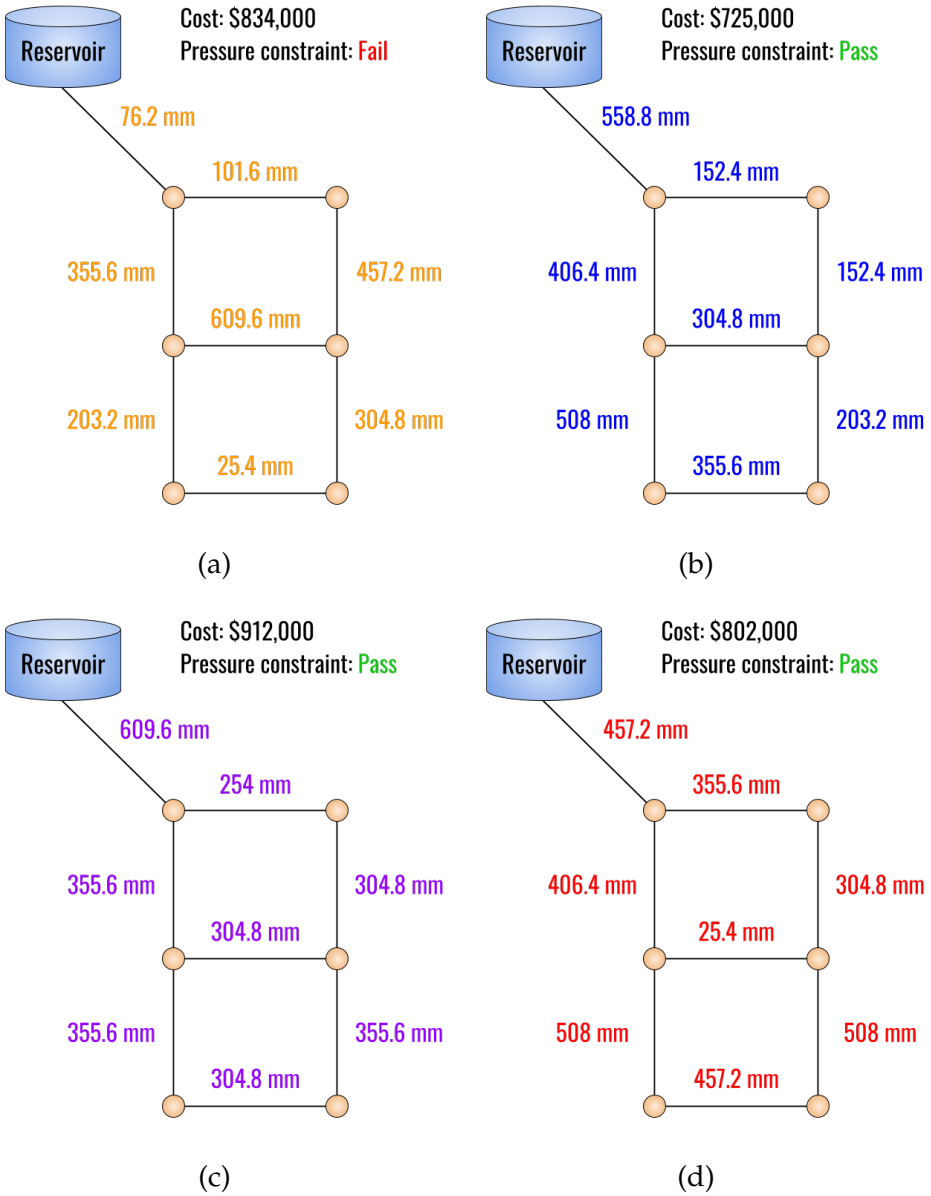


Figure 3.3.: Selection

3.2.3. Crossover

The combination of the least-cost parents into children networks is called the crossover (or mating) function. This author's implementation uses a single-point crossover function: a pipe index number is randomly selected; Parent 1 supplies the pipe diameters up to this index, while Parent 2 supplies the rest of the diameters.

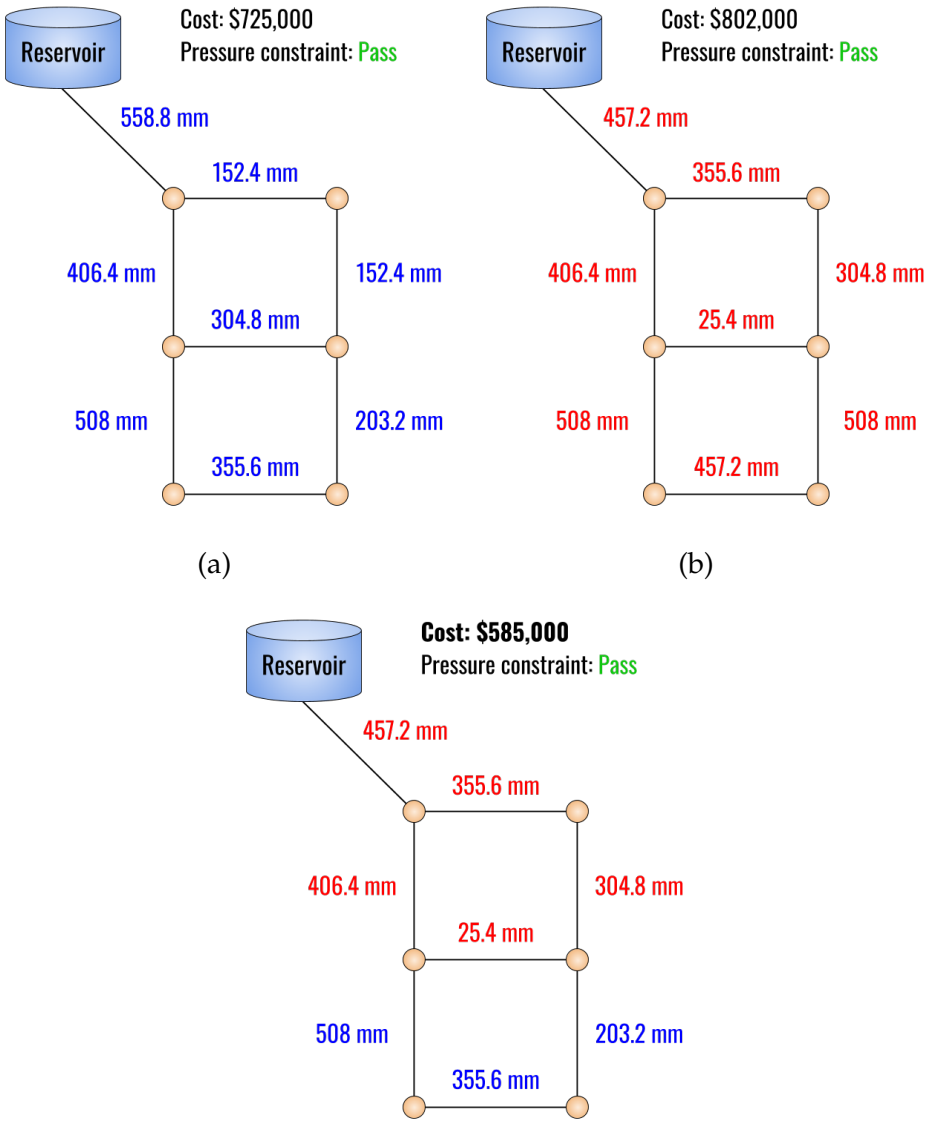


Figure 3.4.: Crossover

In this example, the mating (crossover) of the 2 fittest parents from the previous generation has produced a child network that costs less than either parent, while still satisfying network constraints. Thus, by combining the best 'genes' from each parent, the genetic algorithm has evolved towards a more optimal solution in this generation.

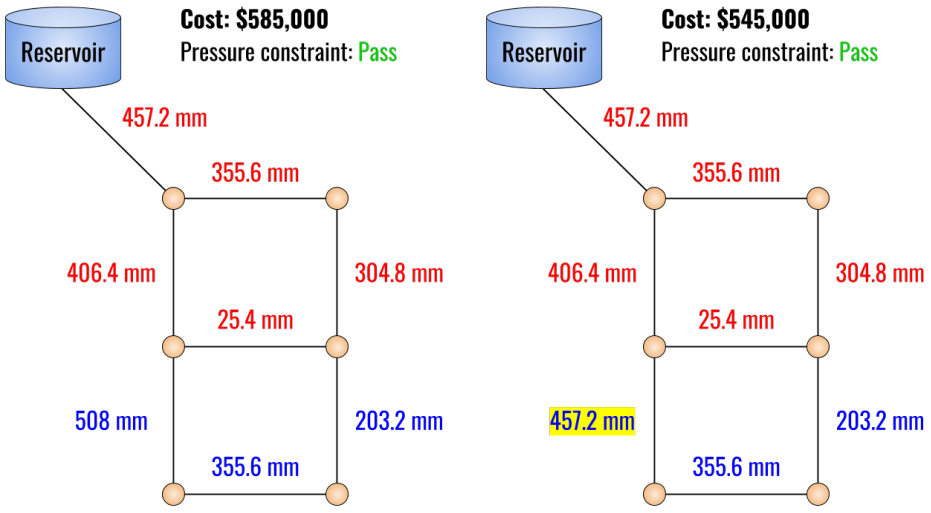
Indeed, it is just as probable that poor genes from both parents are selected, and the resultant child is less optimal than both parents. This author's algorithm addresses this by repopulating the parents and their children in the next generation, together with more randomly generated networks to fill up the original population size. Hence, any children worse off than either parent will not be selected for crossover, and will be discarded in the next generation.

3.2.4. Mutation

Finally, a very small number of pipes (relative to the network size) in the child network are randomly selected for mutation: its diameter is discarded and a new diameter, randomly sampled, is set. By introducing genetic diversity, the mutation function helps to prevent the population from being trapped in local minima.

For example, the pipe on the lower left has a diameter of 508 mm in both parent networks. Hence, while the crossover function decreased the cost of the other pipes, it is unable to reduce the diameter of this pipe by combining the parents' genes. As such, while a more optimal solution exists, it will not be reachable by crossover alone.

By introducing random mutations in the network, the algorithm can 'tunnel' from a local minima to a more optimal solution in the search space. As with the crossover function, the mutation function can also work unfavourably, i.e. if it mutates into a more expensive or infeasible configuration. This is again addressed by repopulating the parent networks for selection against their children in the next generation.



(a) (b)

Figure 3.5.: Mutation

3.2.5. Iteration and Termination

After a large number of generations, the best solution found is expected to evolve towards the global optimum. For large networks, whose solution space cannot be exhaustively searched, it is impossible to test if the genetic algorithm has reached the exact least-cost feasible configuration. However, the genetic algorithm can be expected to approach this global minimum after a large number of generations.

3.3. Results

3.3.1. Two-Loop Network (example theoretical network)

As introduced above, the search space for the Two-Loop Network is:

$$|\mathbb{S}| = N_d^{N_p} = 14^8 = 1.49 \times 10^9 \tag{3.4}$$

The pressure requirement is 30 m for all demand nodes.

The network and the table of pipe prices are entered into the Python program written by this author:

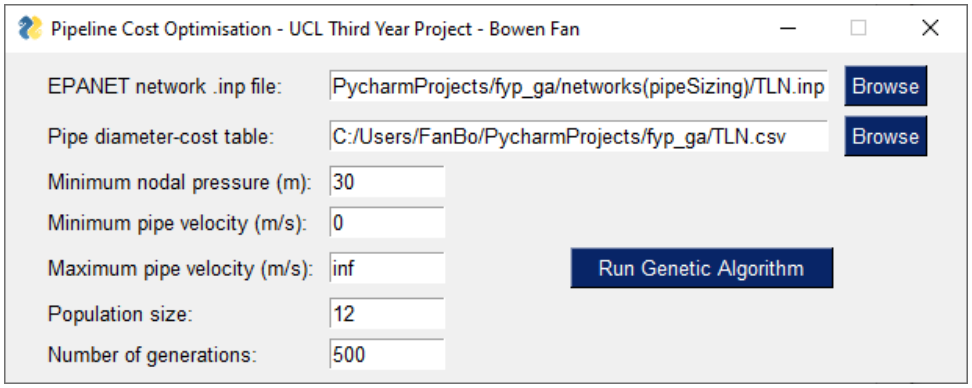


Figure 3.6.: Genetic algorithm program interface

The genetic algorithm is instructed to run for 500 generations. The least-cost feasible network found is recorded and plotted for each generation, as shown in Figure 3.7. The computation time for 500 generations is about 40 seconds.

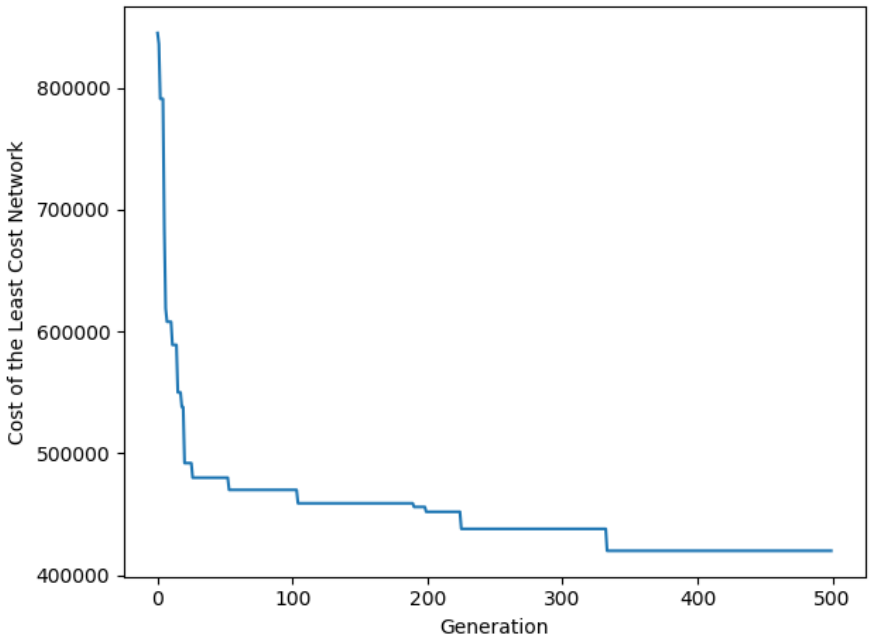


Figure 3.7.: TLN: Solution cost against generations run

After 333 generations, the algorithm has converged to an optimal configuration and has stagnated for the remaining generations. The least-cost feasible pipe configuration found in this run of the genetic algorithm is:

Solution configuration for $420,000 least-cost feasible network								
Pipe ID	1	2	3	4	5	6	7	8
Diameter (mm)	508	254	406.4	25.4	355.6	254	254	25.4
Cost per m ($)	170	32	90	2	60	32	32	2

Table 3.1.: TLN: Best solution found by GA

It is pertinent to note, however, that there is no guarantee for the genetic algorithm to converge to the optimal solution within a fixed number of generations. Sometimes, the algorithm can stagnate at a local minima, unable to 'jump' over to a lower cost feasible configuration. To illustrate this, 10 independent runs of the genetic algorithm were conducted, and their results are presented in Table 3.2 below. Run 1 is the original run shown above. All runs were set to terminate after 500 generations, using a population size of 12.

Runs 1, 4, and 9 managed to converge to the optimal solution, and the solution configuration from Runs 1 and 9 are identical. Comparing the diameter configurations

Run No.	P1	P2	P3	P4	P5	P6	P7	P8	Cost ($'000s)
1	508	254	406.4	25.4	355.6	254	254	25.4	420
2	508	152.4	457.2	254	355.6	254	50.8	25.4	447
3	508	203.2	406.4	355.6	355.6	25.4	203.2	254	460
4	457.2	355.6	355.6	25.4	355.6	152.4	355.6	254	420
5	457.2	203.2	457.2	355.6	355.6	25.4	25.4	304.8	457
6	457.2	304.8	406.4	25.4	406.4	254	254	25.4	428
7	457.2	203.2	457.2	203.2	406.4	254	101.6	50.8	444
8	508	203.2	457.2	203.2	355.6	254	101.6	25.4	451
9	508	254	406.4	25.4	355.6	254	254	25.4	420
10	457.2	304.8	406.4	25.4	406.4	254	254	25.4	428

Table 3.2.: TLN: Result of 10 runs tabulating diameter (mm) for each pipe P1-8

from Solution 1 and 4, it is seen that they are not similar (Figures 3.8 and 3.9). Only the diameters of pipes 4 and 5 remain the same, while the diameter selection of all other pipes were very different. This demonstrates that multiple feasible combinations can exist even for such a simple network with 8 pipes.

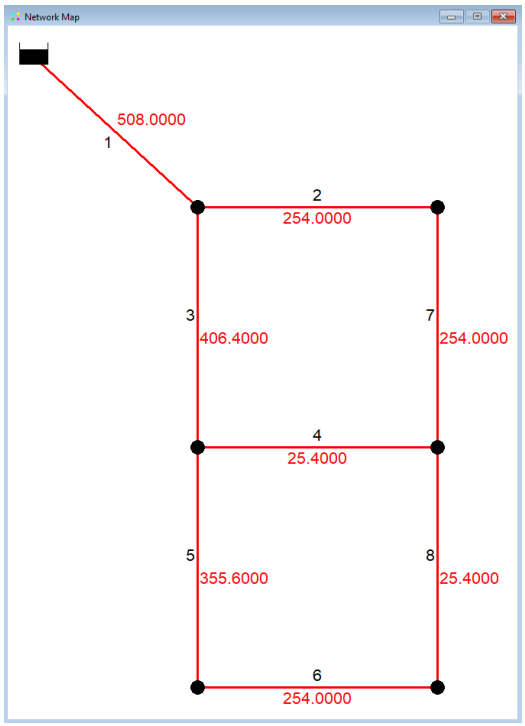


Figure 3.8.: TLN: Run 1 solution

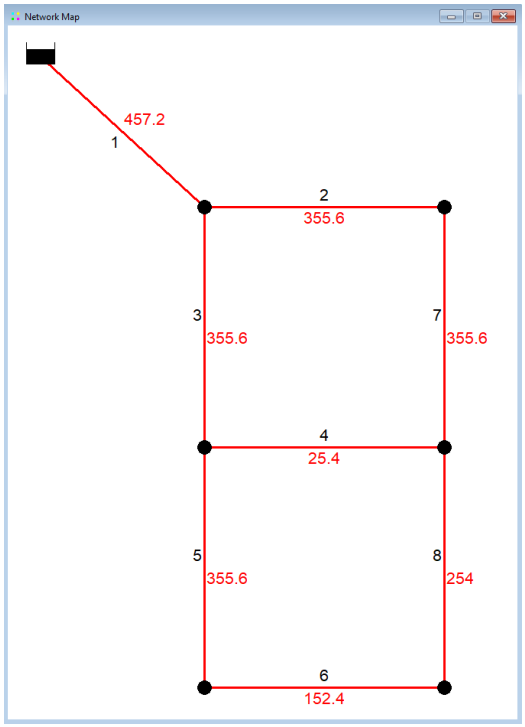


Figure 3.9.: TLN: Run 4 solution

3.3.2. Hanoi Network (medium real-world network)

The Hanoi Network (HAN) (see Figure 3.11) is a skeletonized network model of the water distribution system in Hanoi, Vietnam. It consists of 31 demand nodes, 34 pipes, and 1 reservoir. Again, the pressure requirement is 30 m for all nodes.

There are 6 commercially available pipe diameters to choose from, hence the search space has size:

$$|\mathbb{S}| = N_d^{N_p} = 6^{34} = 2.87 \times 10^{26} \tag{3.5}$$

12 recorded runs of the genetic algorithm were performed on the Hanoi Network. Various numbers of generations and population sizes were used (see Table 3.3). The best solution found costs $6,298,155, with 1500 generations and a population size of 16. The runtime for 500 generations is approximately 65 seconds, and increases linearly with more generations. All generations contains 4 parents, 4 children, and n_r randomly initialised new networks, thus $n_r =$ population size $- 8$.

Cost ($'000s)	Number of generations run	Population size
6298	1500	16
6336	1000	14
6337	3000	16
6404	1500	12
6406	500	12
6426	1000	12
6465	1500	14
6468	500	12
6482	500	14
6491	500	12
6657	1500	16
6713	1000	12

Table 3.3.: HAN: Cost of the best solutions of 12 runs, sorted in ascending cost

For the best solution of $6,298,155, the cost of the network is brought down rapidly, but begins to stagnate after around 800 generations. Afterwards, only marginal improvements were made, until the algorithm was instructed to terminate after 1500 generations (Figure 3.10).

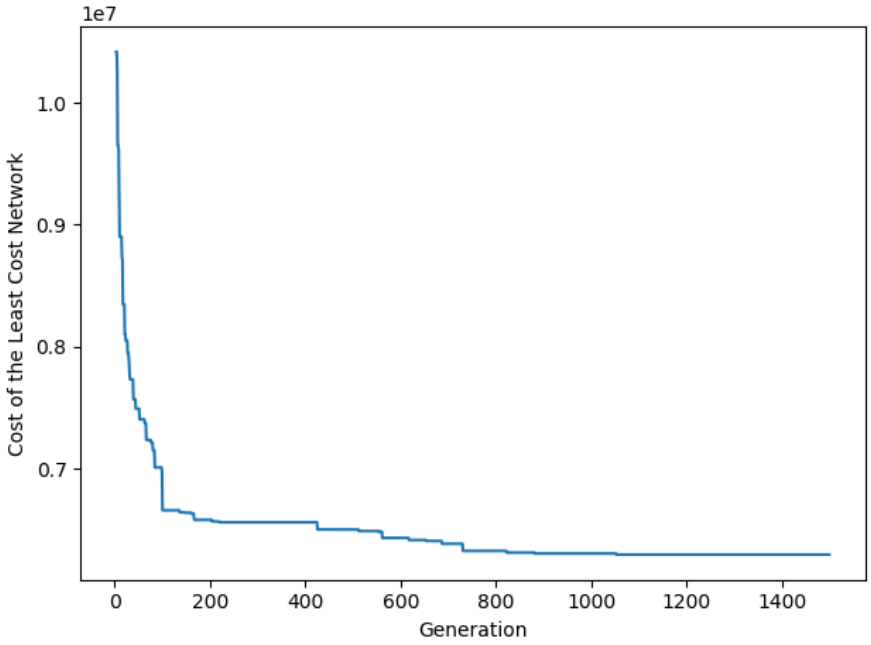


Figure 3.10.: HAN: Solution cost against generations run

The best network configuration discovered is shown graphically in EPANET:

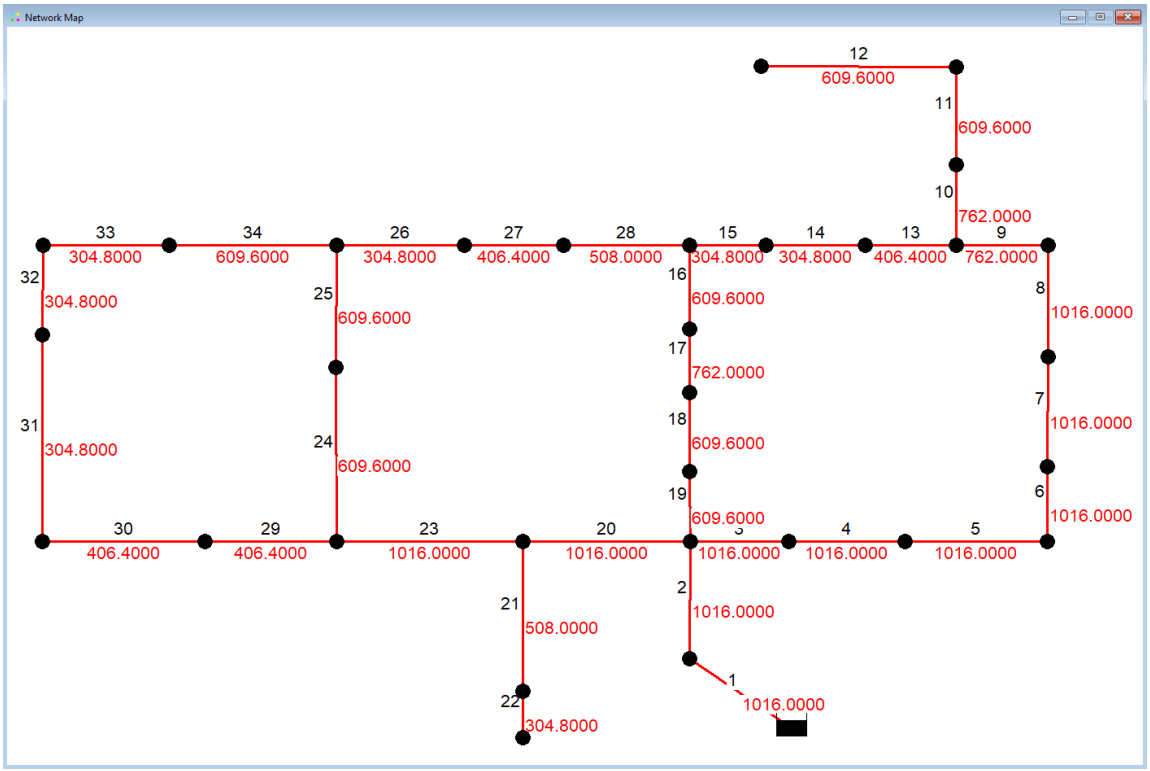


Figure 3.11.: HAN: Diameter configuration of the best solution found

3.3.3. Balerma Irrigation Network (large real-world network)

The Balerma Irrigation Network (BIN) (see Figure 3.13) is a large irrigation water supply system in Balerma, Spain. It contains 434 demand nodes, 454 pipes, and 4 reservoirs. The pressure requirement is 20 m for all nodes.

There are 10 commercially available pipe diameters to choose from, hence the search space has size:

$$|\mathbb{S}| = N_d^{N_p} = 10^{454} \tag{3.6}$$

This is challenging problem that requires a large number of iterations. 5 runs were conducted with 3000, 6000, 12000, 18000 and 24000 generations, all using a population size of 14. The computation time is around 30 mins for every 6000 generations. The best solution found by the GA costs € 2,896,267, discovered by the 24000 generation run. The 12000 generation run actually found a cheaper solution than the 18000 run, showing the stochastic property of GAs.

Number of generations	3000	6000	12000	18000	24000
Cost of best solution found (€'000s)	3607.9	3531.6	2917.1	2969.8	2896.3

Table 3.4.: BIN: Best solution found by each run

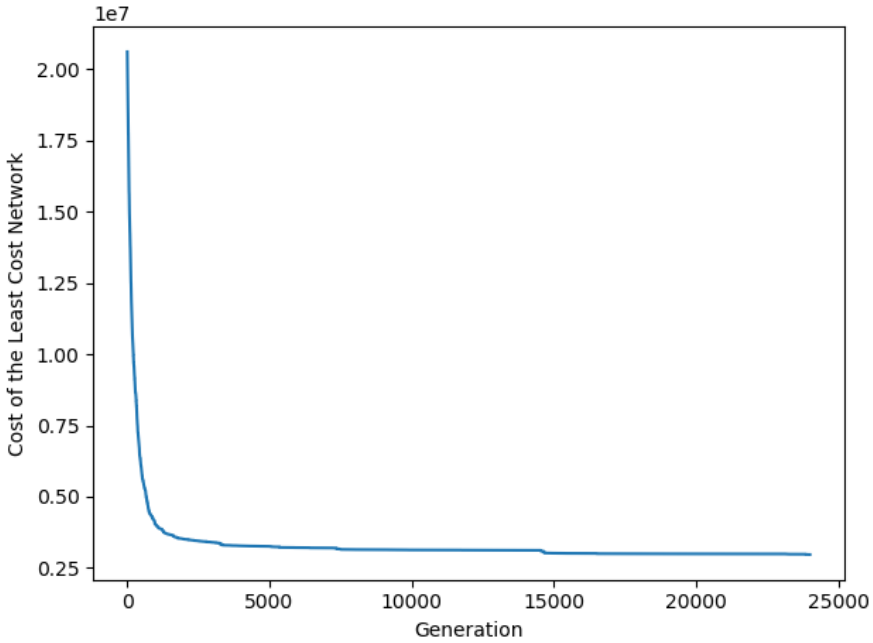


Figure 3.12.: BIN: Solution cost against generations run

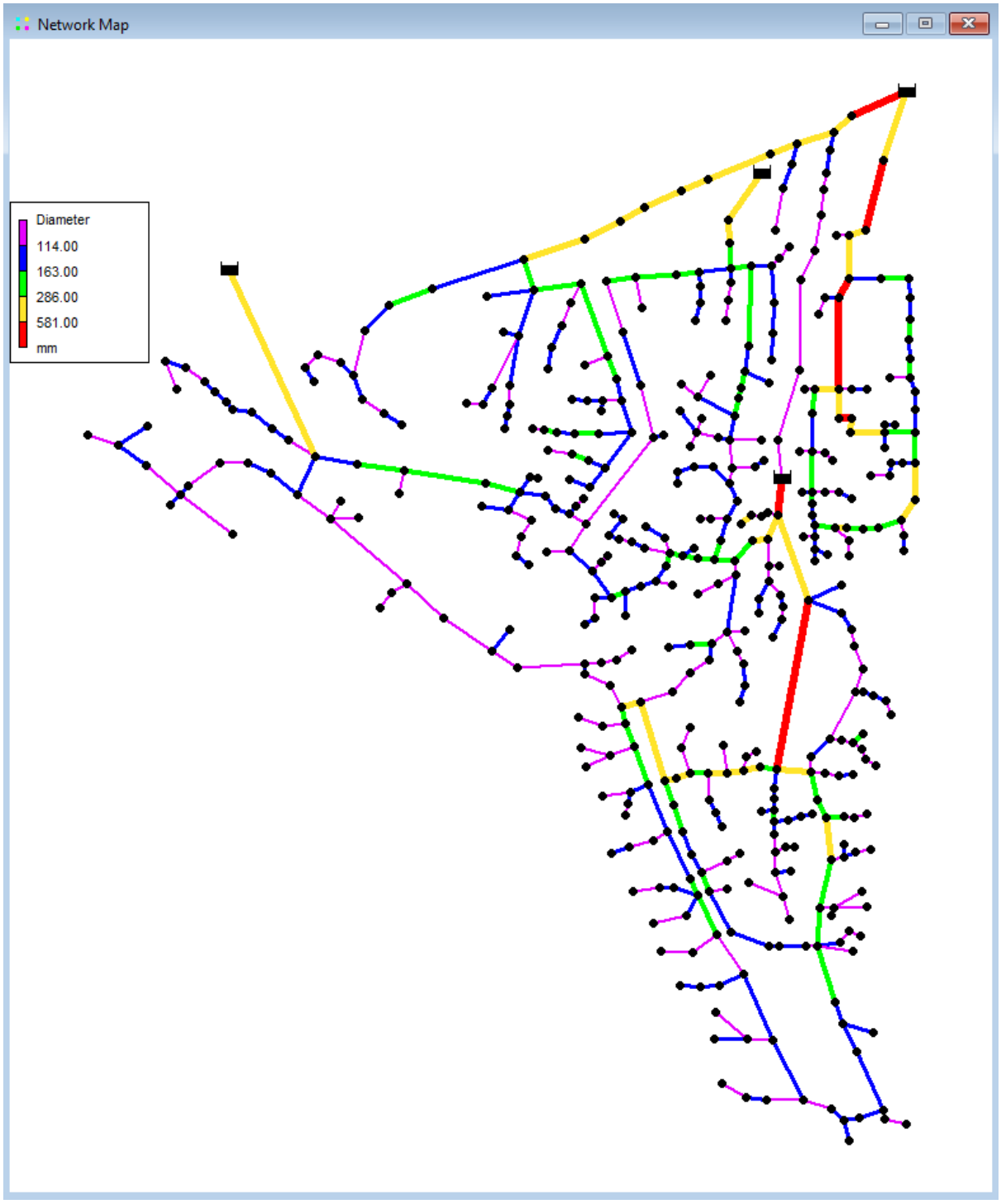


Figure 3.13.: BIN: Diameter configuration of the best solution found

3.3.4. Discussion: Benchmark against literature

A comparison of the best solutions found in this thesis against solutions in literature is given below (Tables 3.5, 3.6, 3.7). For small and medium networks, i.e. TLN and HAN, the performance of this author's algorithm equals or comes close to the performance of the algorithms developed by academic researchers.

Author(s)	Cost of best solution	Algorithm used
This thesis	$ 420,000	Genetic algorithm
Cunha and Ribeiro (2004)	$ 420,000	Tabu search
Eusuff and Lansey (2003)	$ 420,000	Shuffled frog leaping

Table 3.5.: Two-Loop Network: Benchmark of this thesis' solution against literature

Author(s)	Cost of best solution	Algorithm used
This thesis	$ 6,298,155	Genetic algorithm
Zecchin et al. (2006)	$ 6,134,000	Ant colony optimisation
Liong and Atiquzzaman (2004)	$ 6,220,000	Shuffled complex evolution
Reca et al. (2017)	$ 6,208,937	Genetic algorithm

Table 3.6.: Hanoi Network: Benchmark of this thesis' solution against literature

Author(s)	Cost of best solution	Algorithm used
This thesis	€ 2,896,267	Genetic algorithm
Geem (2009)	€ 2,633,000	Particle swarm harmony search
Reca et al. (2008)	€ 3,298,268	Simulated annealing with Tabu search
Sadollah et al. (2014)	€ 2,306,000	Water cycle algorithm (GA variant)

Table 3.7.: Balerma Network: Benchmark of this thesis' solution against literature

However, for the large Balerma Network, this author's algorithm is either not efficient enough, or not sufficiently pushed to its limits by computing an even greater number of generations on more powerful hardware. Using their novel "water cycle algorithm", essentially a genetic algorithm variant, Sadollah et al. (2014) found a solution that is 20% cheaper than the best solution in this thesis. This thesis' solution is however superior to that found by Reca et al. (2008)'s algorithm (simulated annealing with tabu search).

It is noticed that this author's GA has a tendency to stagnate after a large number of generations. Further work will have to be done to alter the algorithm's behaviour when stagnation is detected. For example, the algorithm can be instructed to increase mutation rates when the cost is not lowered after a threshold number of generations.

4. Optimisation of Buffer Tank Placement

4.1. Demand Variations in Real-World Networks

In real-world water distribution systems, water demand varies greatly. Demand vary both on a seasonal and daily basis. Seasonally, demand in spring and summer are generally higher than in autumn and winter. The daily demand pattern usually consists of two demand peaks, in the morning and evening. This is known as a diurnal demand pattern, and is representative of domestic water usage (Carragher et al., 2012).

From a direct measurement of water usage by a residential area in Queensland, Australia, the following diurnal demand pattern is observed (plotted by this author using data from Umapathi et al. (2012)).

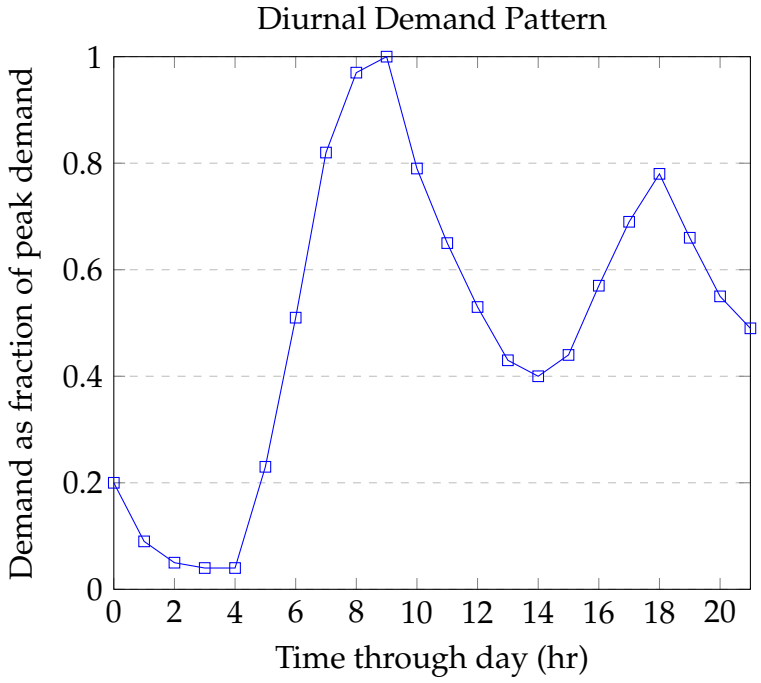


Figure 4.1.: A typical domestic water demand pattern

Water distribution systems must be designed to handle peak daily demands within the peak season. They must furthermore be buffered for fire flow, wherein large volumes of water are drawn for firefighting. Generally, to address the daily imbalance in demand, combinations of the following methods are used (Mays, 1999):

- Demand-balancing tanks can be added at optimal locations in the network. They are filled during off-peak times, and release the stored water into the network during peak demand.
- Pumps can be added upstream, near reservoirs and treatment plants. Thus, hydraulic head is added into the WDS. By scheduling the pumps to run only during peak demand periods, pressures can be increased during peak demand without causing excessive pressures during off-peak times.
- Larger pipes can be used throughout the network to cope with the flow requirements. In the previous chapter, an optimisation method for pipe sizing has been discussed.

The network must be designed such that nodal pressures during peak demand are satisfied, but without causing excessive pressures during off-peak demand, which may compromise physical integrity, e.g. increased risk of pipe bursts. While pumps can be added to balance pressures, the pumps require an active control system for scheduling; moreover, they consume significant energy expenditure and are more prone to mechanical failure.

Demand-balancing tanks are a passive method that can also achieve the same objective. They are advantageous in that they require no control system nor energy expenditure. However, for a tank to have the greatest balancing effect, it must be placed at an optimal location in the network.

Thus, a novel method to determine the optimal locations in the network for adding demand-balancing tanks (also called buffer tanks) has been developed.

4.2. Methodology

Unlike the pipe diameter optimisation problem, whose search space increases exponentially with network size, the search space for the optimal buffer tank location increases only linearly with the size of the network: In a network with n nodes, there are also n possible node locations to attach a buffer tank.

Thus, for the optimisation of buffer tank locations, an exhaustive search is possible.

This author has developed a novel algorithm which attaches a buffer tank successively to each node, then evaluates the effect on network pressures. This algorithm is also implemented as a Python program that calls the EPANET library to solve a network's hydraulics.

An elaboration of the methodology of this algorithm is given below. The Two-Loop Network is again used as an example. The typical diurnal demand pattern in Figure 4.1 is applied to all demand nodes.

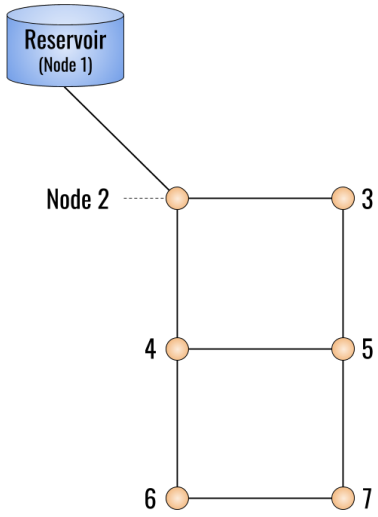


Figure 4.2.: TLN: Reservoir and demand node IDs

1. First, the program imports the network and calls EPANET to retrieve the number of nodes, the x-y coordinates and elevation of each node. The list of elevations for the Two-Loop Network is given in Table 4.1.

Node IDs	Elevation (m)
1 (Reservoir)	210
2	150
3	160
4	155
5	150
6	165
7	160

Table 4.1.: TLN: Elevations of reservoir and demand nodes

2. Next, EPANET is called to solve the network hydraulics for all nodes across all timesteps. This author's algorithm stores the pressure value for each node at each timestep in a Python array, represented in Table 4.2 (truncated at 4 hr intervals).

	Pressure (m) at each node, at the corresponding timestep						
Time	N2	N3	N4	N5	N6	N7	Network Average
1 AM	59.92	49.87	54.87	59.78	44.83	49.78	53.17
5 AM	59.56	49.24	54.25	58.74	44.06	48.74	52.43
9 AM	53.25	38.42	43.66	40.79	30.66	30.79	39.59
1 PM	58.59	47.57	52.62	55.97	42.00	45.98	50.45
5 PM	56.60	44.17	49.29	50.34	37.79	40.34	46.42
9 PM	58.20	46.91	51.97	54.87	41.17	44.87	49.67
	Time-averaged node pressure						
	57.77	46.18	51.26	53.67	40.27	43.67	
Minimum average network pressure (at 9 am):							39.59

Table 4.2.: TLN: Pressure at each node at each timestep

3. Using the pressure array, the average network pressure for each timestep is evaluated. The minimum average network pressure is recorded. Referring to Table 4.2, the minimum average network pressure is 39.59 m, occurring at 9 am. As expected, the minimum average network pressure occurs at the time of maximum demand (see Figure 4.1).

4. For each node, the average of its pressures over the 24 hour period is evaluated. This is the time-average nodal pressure, $\bar{P}_{i,node}$.

   $E^*_{i,tank}$, the optimal elevation for a buffer tank attached at a given demand node i, is found by experiment to be:

$$E^*_{i,tank} \approx E_{i,node} + \bar{P}_{i,node} \tag{4.1}$$

   Where $E_{i,node}$ is the elevation of node i. This relationship will be proved in Subsection 4.2.1.

5. Next, a subroutine calls the EPANET library to edit the network: a tank is added to the network and attached to Demand Node 1. The pipe linking the buffer tank to the node is modelled as very short and smooth (a very high Hazen-William's

coefficient is used). This is to approximate the tank being placed directly above the node.

The elevation of the tank is set according to Equation 4.1. Due to a limitation with EPANET's Demand-Driven Analysis solver, it is necessary to ensure that the water level in the buffer tank is kept low. Hence, a large diameter of 100 m is set for the tank, so that it can store the buffer volume without a significant increase in water level. This issue is discussed in Subsection 4.2.2.

6. After the buffer tank is attached to Node 1, EPANET is called to solve network hydraulics through all timesteps again. The minimum average network pressure (average network pressure during demand peak) is evaluated and recorded. Thus, the algorithm repeats steps 2 to 5 each time a tank is added to the next node.
7. The buffer tank and its pipe are deleted. The network is reset.
8. A buffer tank is now added to Demand Node 2. Thus, steps 2 to 7 are repeated until every potential buffer tank location has been tested. Each location's corresponding minimum average network pressure is recorded.
9. Finally, the program outputs an array listing the tank locations and the corresponding network pressure. This array is sorted with the best locations listed first. For the TLN, Node 6 was found to be the best location (see Figure 4.3).

4.2.1. Discussion: Experimental Proof for Optimal Elevation

A trial-and-error experiment was conducted using the EPANET engine, to determine the optimal elevation for a buffer tank attached at any demand node i.

Using the Two-Loop Network, the best-obtainable minimum average network pressure was evaluated for each of these elevation equations:

$$E_{i,tank} = E_{i,node} \tag{4.2}$$

$$E_{i,tank} = E_{i,node} + 0.25\,\bar{P}_{i,node} \tag{4.3}$$

$$E_{i,tank} = E_{i,node} + 0.50\,\bar{P}_{i,node} \tag{4.4}$$

$$E_{i,tank} = E_{i,node} + 0.75\,\bar{P}_{i,node} \tag{4.5}$$

$$E_{i,tank} = E_{i,node} + \bar{P}_{i,node} \tag{4.6}$$

The following results were obtained:

Equation used to set tank elevation	Best-obtainable min. avg. network pressure (m)
Original network, no tank	39.59
$E_{i,tank} = E_{i,node}$	30.03
$E_{i,tank} = E_{i,node} + 0.25\,\bar{P}_{i,node}$	32.93
$E_{i,tank} = E_{i,node} + 0.50\,\bar{P}_{i,node}$	36.52
$E_{i,tank} = E_{i,node} + 0.75\,\bar{P}_{i,node}$	40.43
$E_{i,tank} = E_{i,node} + \bar{P}_{i,node}$	46.47
$E_{i,tank} = 1000m$	836.35 (Incorrect due to DDA: see Section 4.2.2)

Table 4.3.: TLN: Peak demand avg. pressure depends on buffer tank elevation

From the experiment results, it can be seen that setting $E_{i,tank} = E_{i,node}$ actually causes the buffer tank to act as a sink in the system. Thus, it causes the average network pressure during peak demand to be even lower than in the original network with no buffer tank.

It is shown that, by setting the elevation of the buffer tank at each node at $E_{i,tank} = E_{i,node} + \bar{P}_{i,node}$, the best possible peak-demand network pressure of 46.47 m is obtained.

Figure 4.3 shows the calculated optimal tank elevation values for the tank attached to each node.

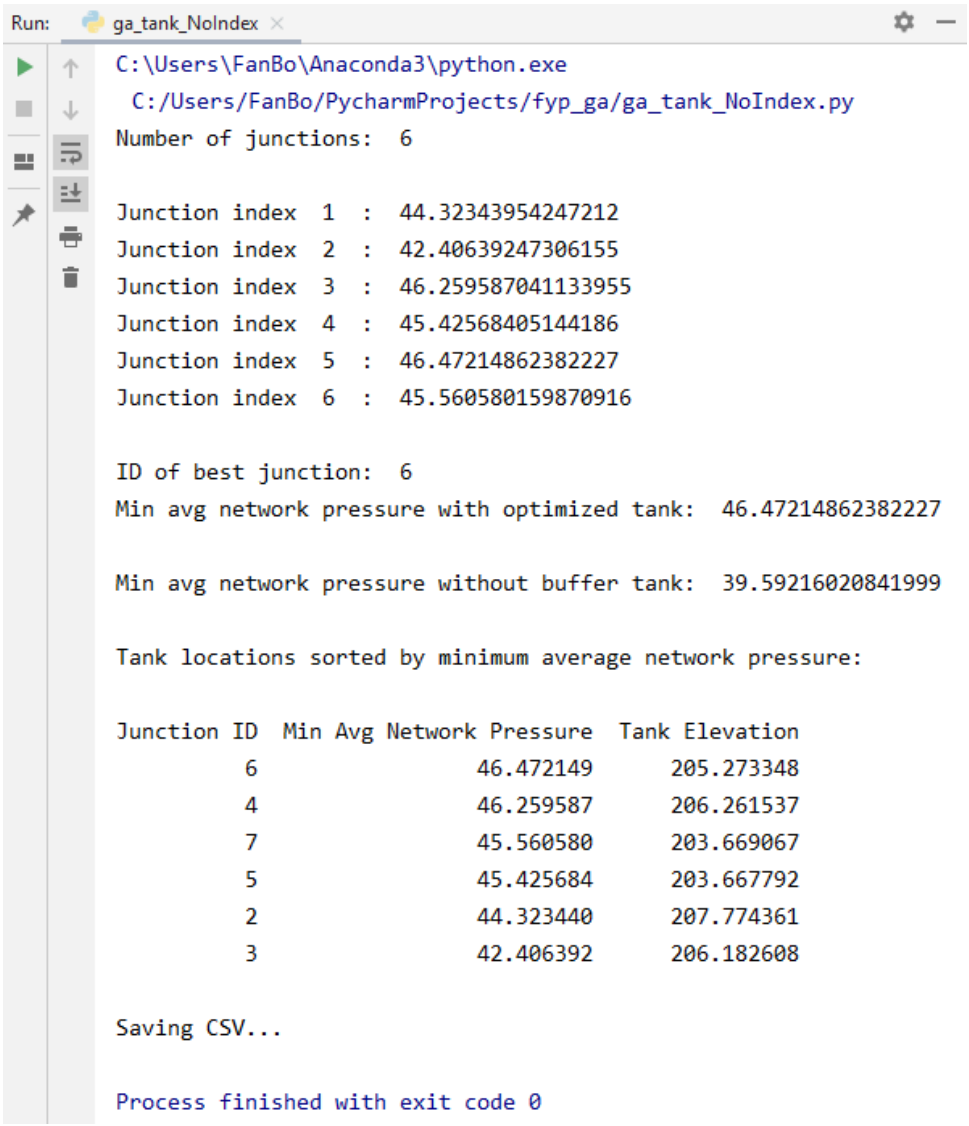


Figure 4.3.: TLN: Screenshot showing program output, ranking the tank locations

4.2.2. Discussion: Limitation in EPANET'S DDA Engine

Another experiment was conducted to study how EPANET models tanks of different diameters. Tanks of various diameters (other properties held constant) were added to the TLN and their effect studied (see Table 4.4).

The result when using a tank with diameter 0.000001 m is clearly incorrect: theoretically, to contain just 1 m^3 of water, the tank must be filled up to over 1 km. This is impossible because the head of the reservoir is only 210 m. Thus, adding a buffer tank of this size should have no effect at all on the network. It should certainly not raise network pressures. Using a 1 m diameter tank yields incorrect results too, with EPANET showing that the 210 m reservoir can top up the tank to over 700 m (Figure 4.4).

The cause of this incorrect result is likely due to EPANET's Demand-Driven Analysis.

Tank Diameter (m)	Best-obtainable min. avg. network pressure (m)
Original network, no tank	39.59
100	46.47
0.000001	46.35 (Incorrect)
1	46.35 (Incorrect)

Table 4.4.: TLN: Incorrect results

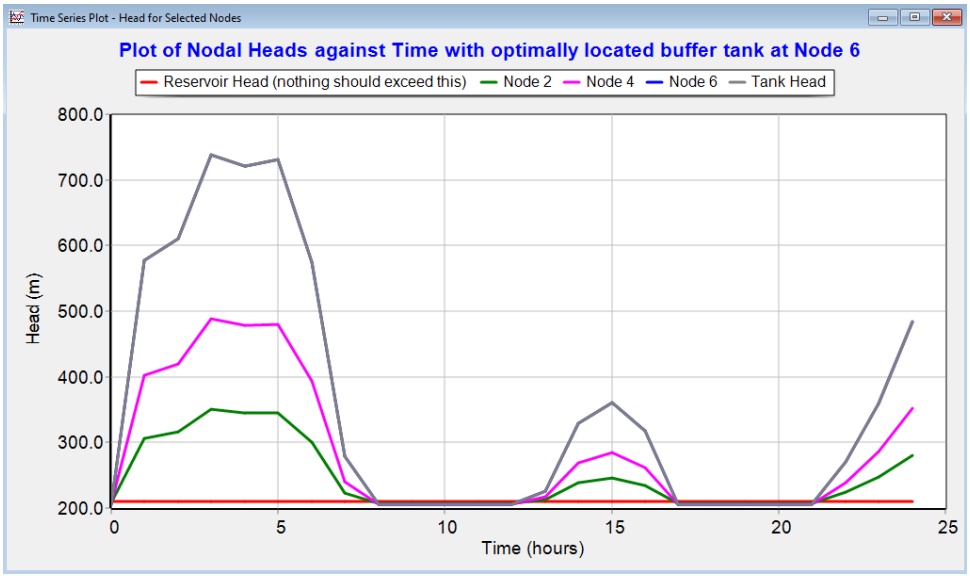


Figure 4.4.: TLN: Small 1 m tank causes impossible hydraulic heads

As explored in Section 2.3, EPANET assumes that nodal demands are always satisfied, then uses flow to calculate pressures. Studying the results, EPANET seems to have modelled the tank as a demand node, thus satisfying its flow regardless of actual pressure.

This specific error is not mentioned in the EPANET documentation, thus its exact mechanism will have to be further researched. Curiously, EPANET did not output any negative pressure warnings, which it usually does if the network is pressure-deficient.

4.3. Results: Modena Network

Modena is a city in northern Italy. The Modena Network (MOD), shown below, is a skeletonized model of the city's WDS. It consists of 268 demand nodes, 317 pipes, and 4 reservoirs at the outer regions of the city. The demand pattern from Figure 4.1 is applied to all nodes.

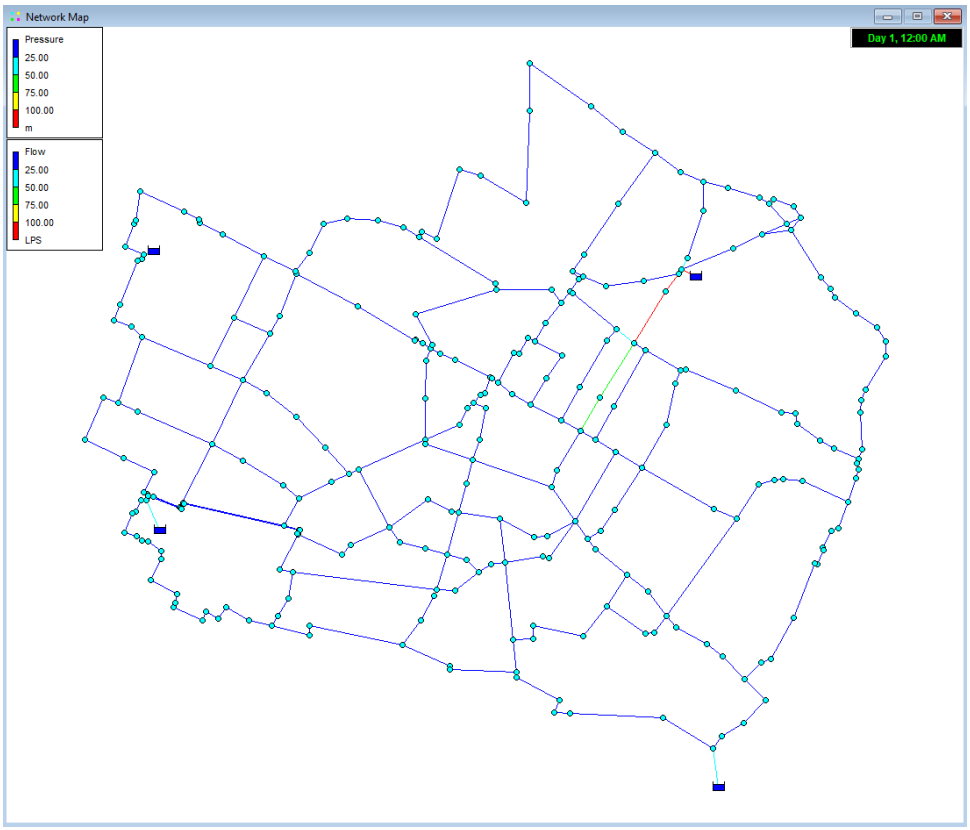


Figure 4.5.: MOD: The Modena Network opened in EPANET

The original network, without any buffer tanks, was first analysed in EPANET to study the pressure distribution within the network at the time of peak demand, 9 am. A frequency profile of the nodal pressures was plotted in EPANET (Figure 4.6).

From the pressure distribution, it can be seen that over 40% of the demand nodes have pressures less than 23 m during peak-demand. A readout from the graph is tabulated in Table 4.6 (2^{nd} column). To visualise the network status, a contour plot for the network at 9 am was drawn in EPANET (Figure 4.7).

Next, the network is imported into this author's Python program (Figure 4.8). The program analyses the network for the optimal buffer tank location, using the algorithm explained in the methodology above. The results found are shwon in Table 4.5, in descending order of the network average pressure during peak-demand.

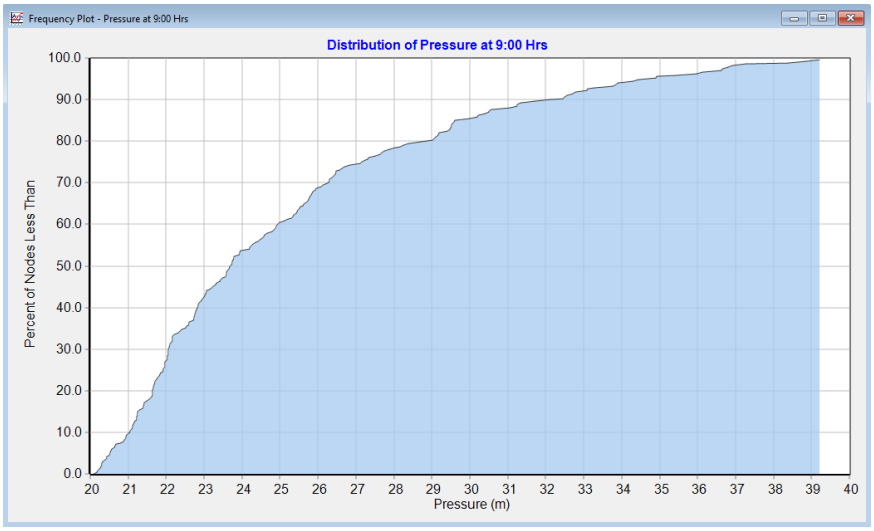


Figure 4.6.: MOD: Frequency distribution of network nodal pressures at 9 am (no tank)

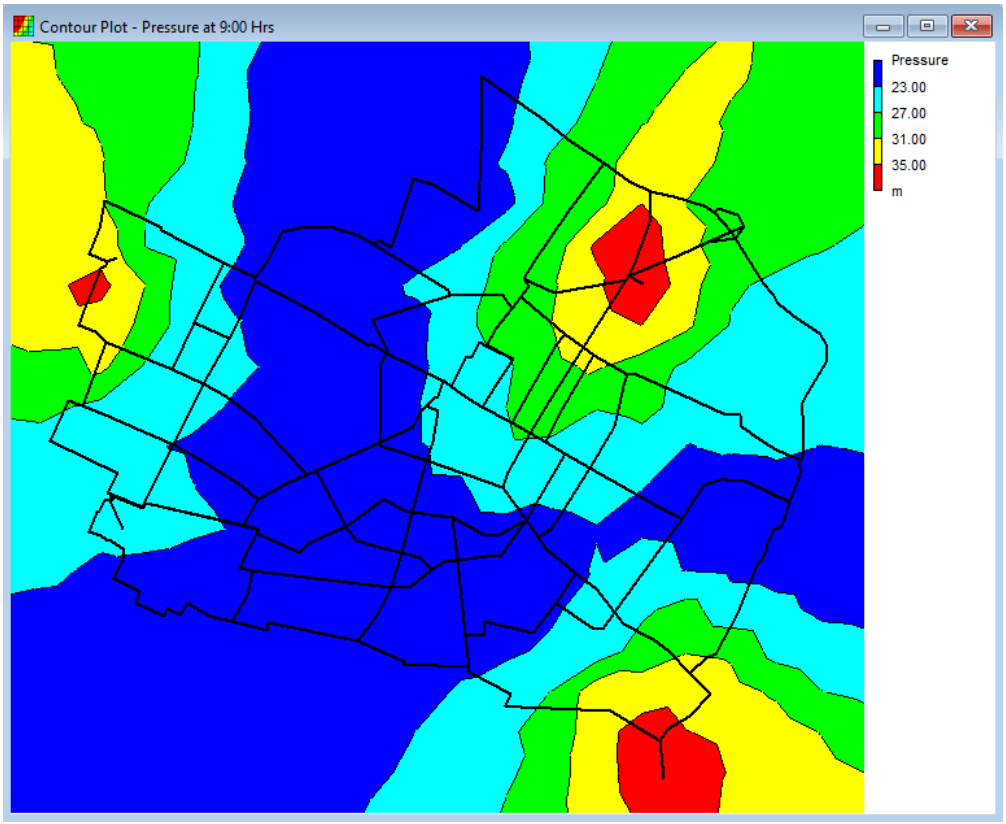


Figure 4.7.: MOD: Contour plot of nodal pressures (no tank)

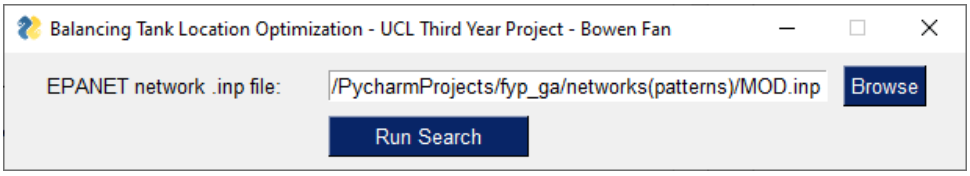


Figure 4.8.: Buffer tank optimisation program interface

Node ID	Average Network Pressure at 9 am (m)	Tank Elevation (m)
101	27.904	67.86
221	27.898	67.88
139	27.879	67.95
92	27.841	67.89
219	27.752	68.12

Table 4.5.: MOD: List of optimal buffer tank locations (Results truncated to top 5)

Hence, for the purpose of increasing the average network pressure during peak demand, Node 101 is determined by the program to be the most optimal location to add a buffer tank. This result is now verified in EPANET, by manually adding a tank and plotting the network status:

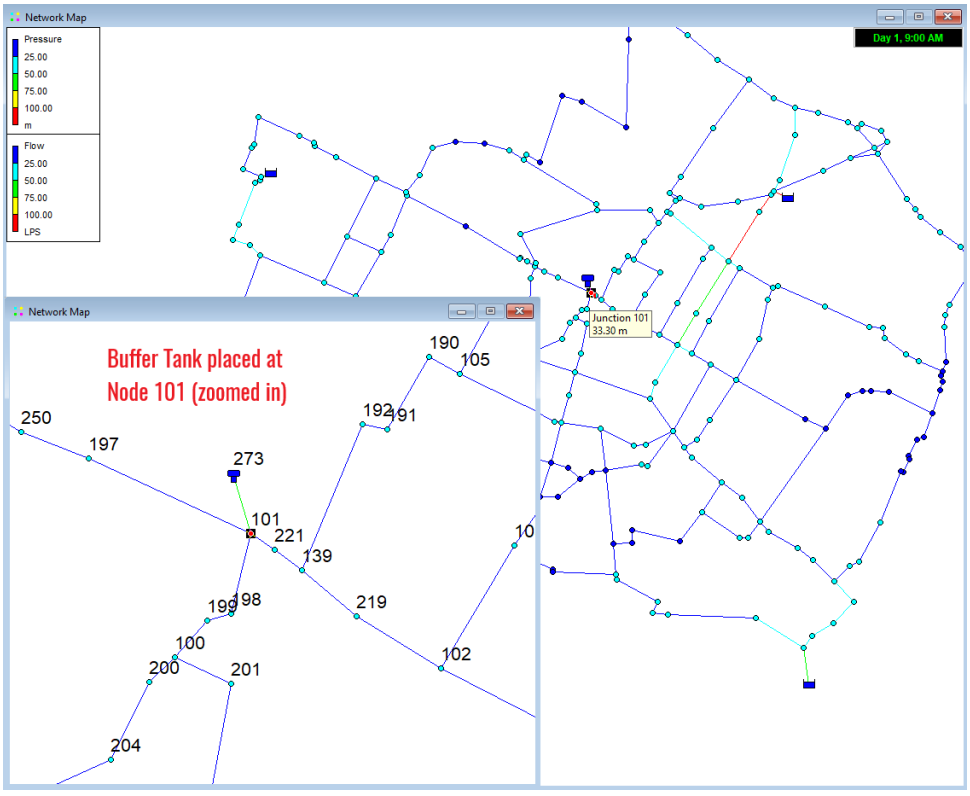


Figure 4.9.: MOD: Location of Node 101, with buffer tank attached

After the buffer tank has been manually added and connected to Node 101, the frequency profile of the network pressures at 9 am is plotted again (Figure 4.10). The pressure distribution shows that the peak-demand network performance has been significantly improved by adding a tank to Node 101, the best location calculated.

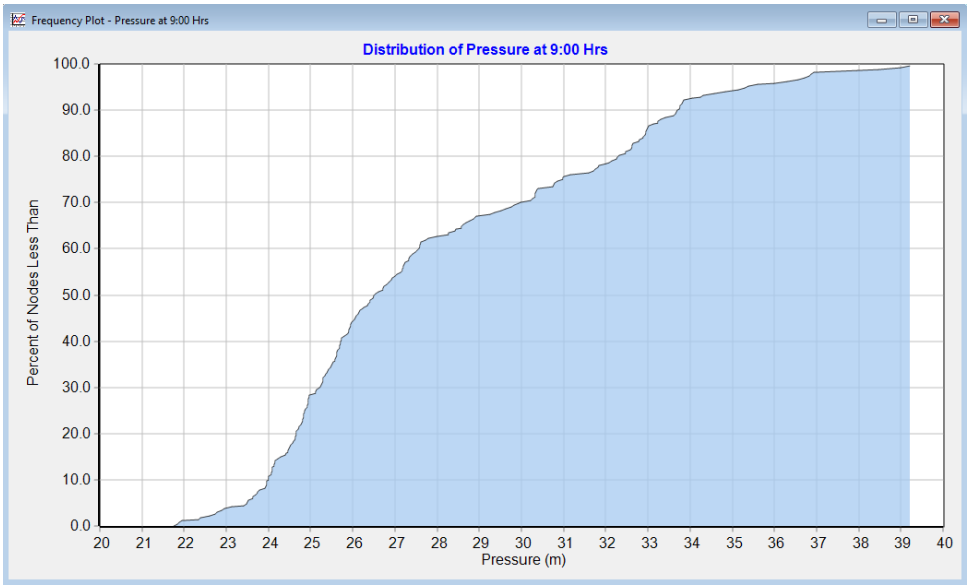


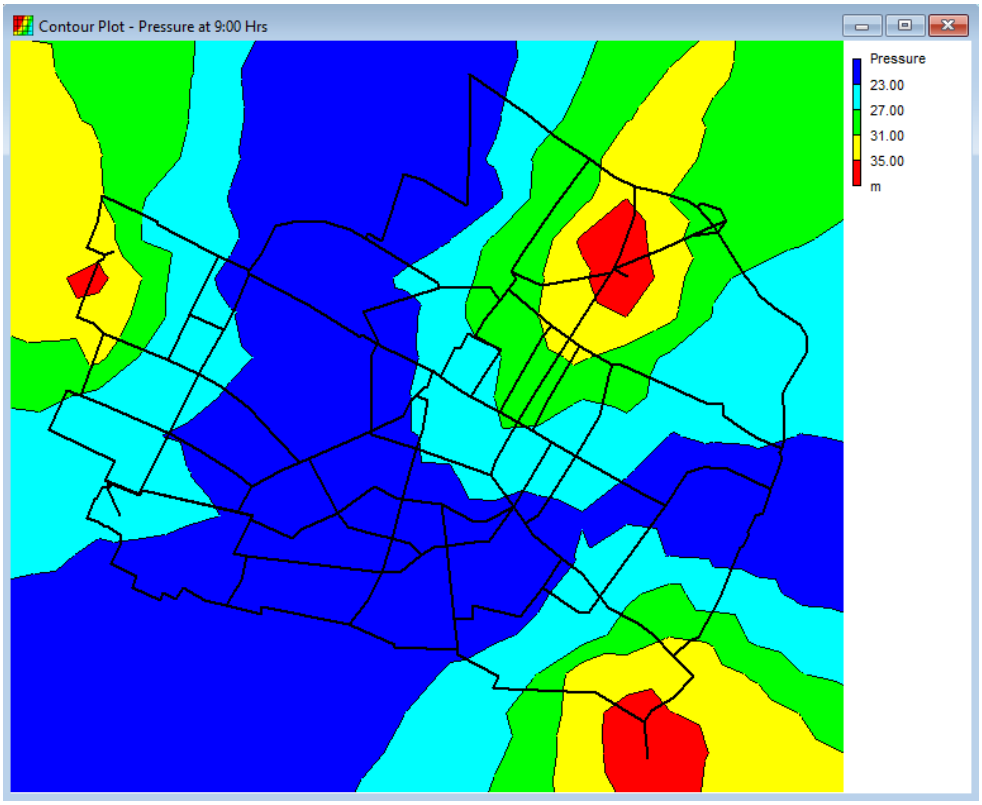
Figure 4.10.: MOD: Distribution of nodal pressures at 9 am (tank at Node 101)

The results after adding the buffer tank to Node 101 is compared to the results obtained on the original network:

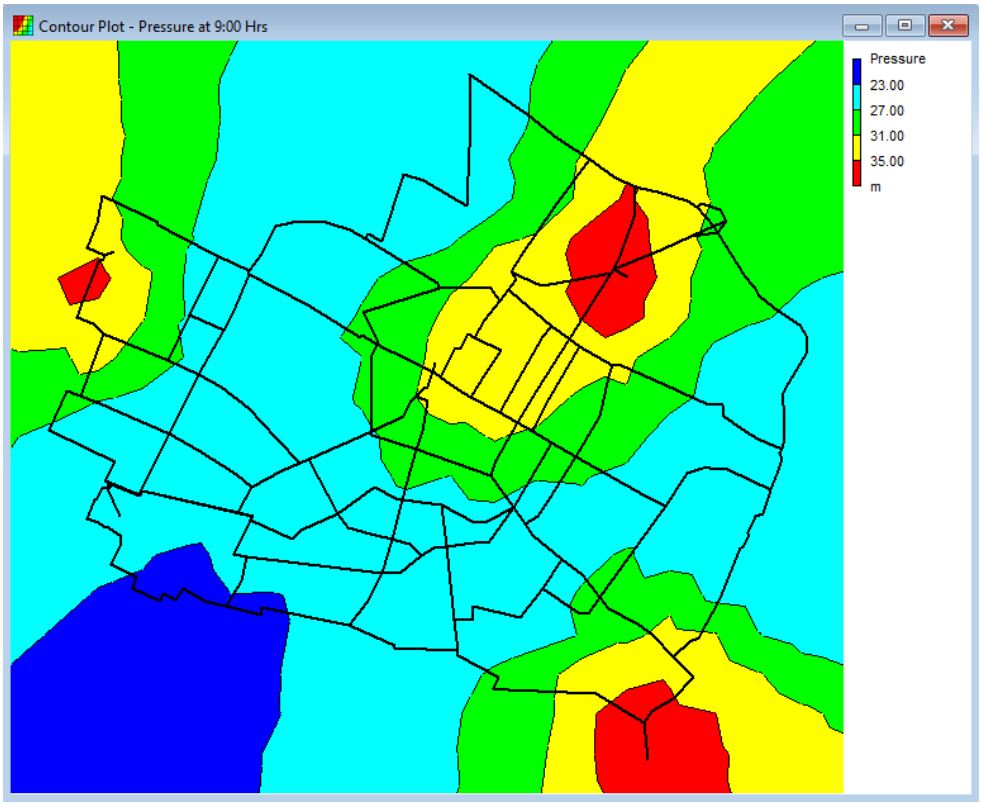
Network pressure distribution at time of peak demand, 9 am		
Pressure range (m)	% of nodes (no tank)	% of nodes (tank at Node 101)
20.1 – 23	42.5	4
23 – 27	32	50
27 – 31	13	21
31 – 35	7.5	19
35 – 39.2	5	6
Avg. network pressure at 9 am	25.13 m	27.90 m

Table 4.6.: MOD: Original network v. improved network with tank at Node 101

The extent of the improvement can be visualised from the before and after contour plots in Figures 4.11 (a) and (b). Note the significantly smaller darker blue area representing nodes with pressure < 23 m.



(a)



(b)

Figure 4.11.: MOD: No tank (a) v. tank at Node 101 (b)

4.3.1. Discussion: Demonstrating the need for an automated search algorithm

It may seem intuitive that, being close to the geographical centre of the network, Node 101 is the best place to add a buffer tank. This seems to agree with the conventional 'rule-of-thumb' described in the literature review section. (Mays, 1999). This section provides a counterexample to this 'rule-of-thumb', demonstrating the need for an automated search algorithm.

Node 201 is located very near Node 101, is of similar elevation, and is also at the centre of the network. Its demand is even higher than that of Node 101 (3.43 and 3.19 LPS respectively). According to conventional guidelines, Node 201 should be an even better candidate due to its higher demand. However, the search algorithm revealed that Node 201 comes in 61st out of the possible 268 node choices.

To investigate this, a buffer tank was manually added to Node 201 in EPANET, and network analysis was again conducted.

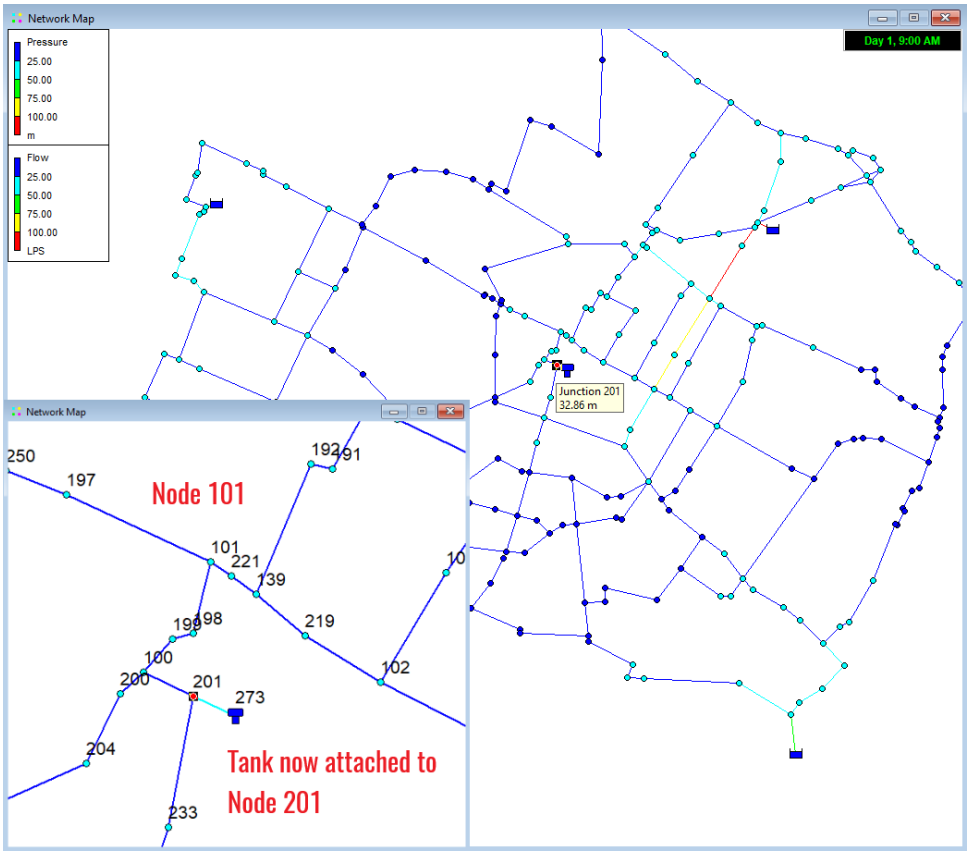


Figure 4.12.: MOD: Relative locations of Nodes 101 and 201

Again, the pressure distribution plot is obtained for the network at 9 am (Figure 4.13). The values are read out and tabulated against the results obtained previously, where

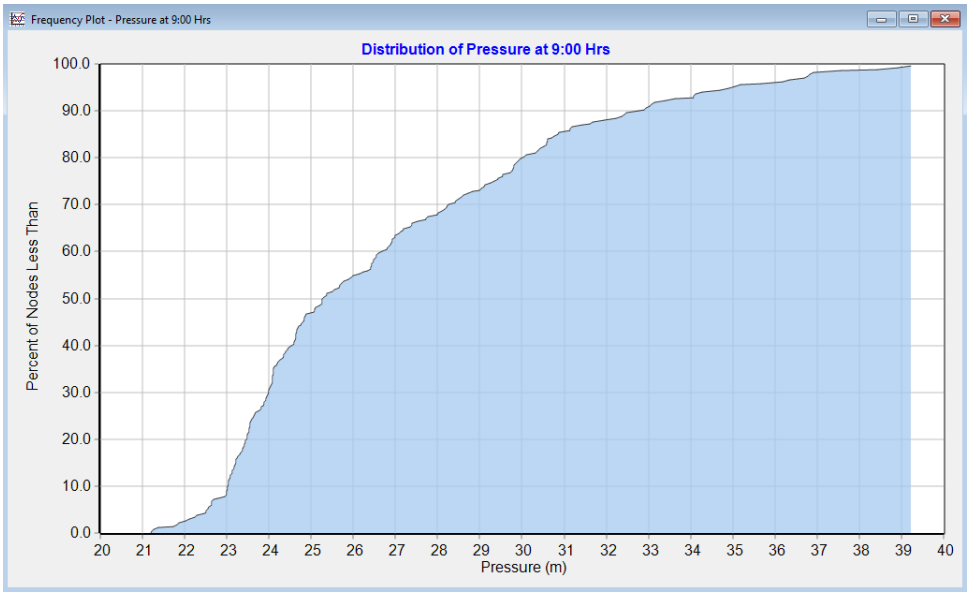


Figure 4.13.: MOD: Distribution of nodal pressures at 9 am (tank at Node 201)

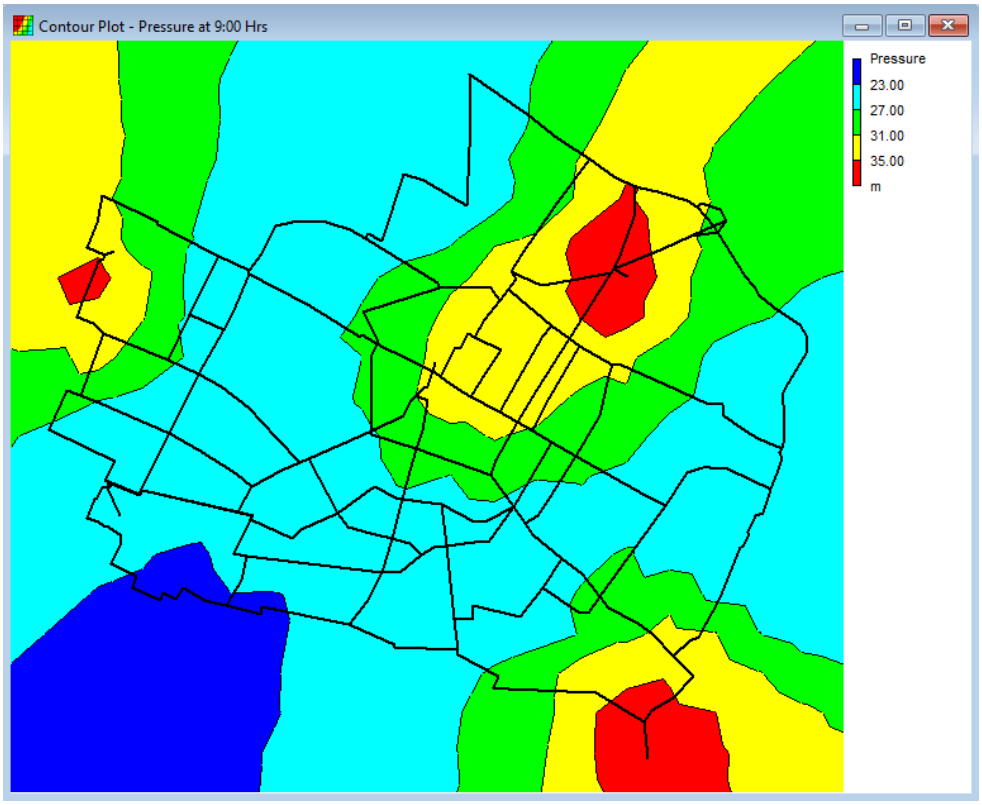
the tank was added at the optimal location of Node 101 (Table 4.7).

Network pressure distribution at time of peak demand, 9 am		
Pressure range (m)	% of nodes (tank at Node 201)	% of nodes (tank at Node 101)
20.1 – 23	10	4
23 – 27	52.5	50
27 – 31	22.5	21
31 – 35	10	19
35 – 39.2	5	6
Avg. network pressure at 9 am	26.71 m	27.90 m

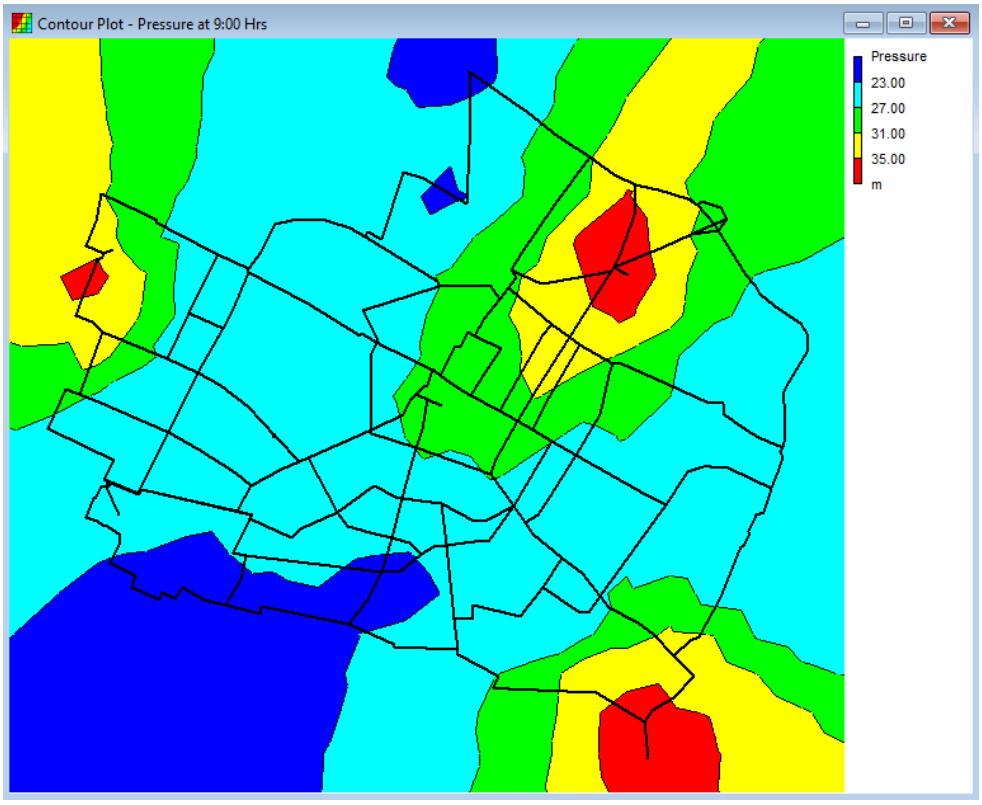
Table 4.7.: MOD: Network with Tank at Node 201 v. Node 101

With a tank added to Node 201, 10% of demand nodes will still experience pressures less than 23 m during peak demand. In contrast, at the optimal location of Node 101, only 4% of demand nodes will experience pressures less than 23 m. Furthermore, for Node 201, the percentage of high pressure nodes ($\geq$31 m) is 15%, while it is 25% for Node 101.

A comparison of the contour plots in Figure 4.14 illustrates the difference. Note the increased darker blue area representing low pressure nodes.



(a)



(b)

Figure 4.14.: MOD: Tank at Node 101 (a) v. tank at Node 201 (b)

5. Conclusions and Further Work

In this project, two algorithms for optimising water distribution systems were developed:

The genetic algorithm for pipeline optimisation performs well for small and medium networks, being able to discover solutions that are near the global optimum. However, for large networks, the algorithm needs to be refined and/or tested on more powerful hardware to determine its efficacy. Due to the COVID-19 UCL closure, it was not possible to push this algorithm to its limit on UCL's more powerful computing facilities.

Beyond the genetic algorithm's performance, further work can be done to add a network reliability factor to the algorithm. Thus, the algorithm would not be fixated on a single objective to minimise cost, but would instead consider a balance of multiple objectives. This will make it more applicable for real-world usage, where pipeline design is a multi-objective decision problem.

The algorithm for optimising buffer tank location is novel. The results of this algorithm were tested manually in EPANET; this confirmed that the algorithm performs well. Additionally, a novel proof was presented for an approximation for the optimal elevation of the tank, given the node to which it is attached. Finally, this study revealed a limitation in EPANET's modelling of water tanks, wherein a tank can incorrectly be topped up to a level beyond the head of the highest reservoir.

In determining a buffer tank's efficacy, the metric used was the average network pressure during peak demand. Further work can be done so that the algorithm scores a buffer tank based not only on the average peak demand network pressure, but also the minimum nodal pressure: while the average pressure gives an indication of the network's health, it is the node with the lowest pressure that will cause problems for end-users. Finally, as buffer tanks are also used to supplement flow for firefighting, this algorithm can also be tested on a demand pattern for fire flow. Indeed, for this purpose, the more applicable metric may be the network's minimum nodal pressure, rather than its average.

The two Python programs written by this author will be made open-source and free for anyone's use. It is hoped that, beyond this project's academic findings, the tools developed will be useful to those designing a WDS, and thus be a tiny contribution for the benefit of WDS community engineers everywhere.

References

Abdy Sayyed, M.A.H., Gupta, R., and Tanyimboh, T.T., 2015. Noniterative application of epanet for pressure dependent modelling of water distribution systems. *Water resources management*, 29(9), pp.3227–3242. Available from: https://doi.org/10.1007/s11269-015-0992-0.

Abunada, M., Trifunović, N., Kennedy, M., and Babel, M., 2014. Optimization and reliability assessment of water distribution networks incorporating demand balancing tanks. *Procedia engineering*, 70, pp.4–13. Available from: https://doi.org/10.1016/j.proeng.2014.02.002.

Aklog, D. and Hosoi, Y., 2017. All-in-one model for designing optimal water distribution pipe networks. *Drinking water engineering and science*, 10(1), pp.33–38. Available from: https://doi.org/10.5194/dwes-10-33-2017.

Ang, W.K. and Jowitt, P.W., 2006. Solution for water distribution systems under pressure-deficient conditions. *Journal of water resources planning and management*, 132(3), pp.175–182. Available from: https://doi.org/10.1061/(asce)0733-9496(2006)132:3(175).

Bohórquez, J., Saldarriaga, J., and Vallejo, D., 2015. Pumping pattern optimization in order to reduce wds operation costs. *Procedia engineering*, 119, pp.1069–1077. Available from: https://doi.org/10.1016/j.proeng.2015.08.936.

Carragher, B.J., Stewart, R.A., and Beal, C.D., 2012. Quantifying the influence of residential water appliance efficiency on average day diurnal demand patterns at an end use level: a precursor to optimised water service infrastructure planning. *Resources, conservation and recycling*, 62, pp.81–90. Available from: https://doi.org/10.1016/j.resconrec.2012.02.008.

Chang, Y., Choi, G., Kim, J., and Byeon, S., 2018. Energy cost optimization for water distribution networks using demand pattern and storage facilities. *Sustainability*, 10(4). Available from: https://doi.org/10.3390/su10041118.

Dong, X.-l., Liu, S.-q., Tao, T., Li, S.-p., and Xin, K.-l., 2012. A comparative study of differential evolution and genetic algorithms for optimizing the design of water distribution systems. *Journal of zhejiang university science a*, 13(9), pp.674–686. Available from: https://doi.org/10.1631/jzus.A1200072.

Geem, Z.W., 2006. Optimal cost design of water distribution networks using harmony search. *Engineering optimization*, 38(3), pp.259–277. Available from: https://doi.org/10.1080/03052150500467430.

Georgescu, A.-M., Georgescu, S.-C., Cosoiu, C.I., Hasegan, L., Anton, A., and Bucur, D.M., 2015. Epanet simulation of control methods for centrifugal pumps operating under variable system demand. *Procedia engineering*, 119, pp.1012–1019. Available from: https://doi.org/10.1016/j.proeng.2015.08.995.

Giustolisi, O. and Laucelli, D., 2011. Water distribution network pressure-driven analysis using the enhanced global gradient algorithm (egga). *Journal of water resources planning and management*, 137(6), pp.498–510. Available from: https://doi.org/10.1061/(asce)wr.1943-5452.0000140.

Gorev, N.B., Kodzhespirova, I.F., and Sivakumar, P., 2016a. Modeling of flow control valves with a nonzero loss coefficient. *Journal of hydraulic engineering*, 142(11). Available from: https://doi.org/10.1061/(asce)hy.1943-7900.0001197.

Gorev, N.B., Kodzhespirova, I.F., and Sivakumar, P., 2016b. Nonunique steady states in water distribution networks with flow control valves. *Journal of hydraulic engineering*, 142(9). Available from: https://doi.org/10.1061/(asce)hy.1943-7900.0001156.

Gorev, N.B., Gorev, V.N., Kodzhespirova, I.F., Shedlovsky, I.A., and Sivakumar, P., 2018. Simulating control valves in water distribution systems as pipes of variable resistance. *Journal of water resources planning and management*, 144(11). Available from: https://doi.org/10.1061/(asce)wr.1943-5452.0001002.

Gorev, N.B., Kodzhespirov, I.F., Kovalenko, Y., Prokhorov, E., and Trapaga, G., 2013. Method to cope with zero flows in newton solvers for water distribution systems. *Journal of hydraulic engineering*, 139(4), pp.456–459. Available from: https://doi.org/10.1061/(asce)hy.1943-7900.0000694.

Gorev, N.B. and Kodzhespirova, I.F., 2013. Noniterative implementation of pressure-dependent demands using the hydraulic analysis engine of epanet 2. *Water resources management*, 27(10), pp.3623–3630. Available from: https://doi.org/10.1007/s11269-013-0369-1.

Gorev, N.B., Kodzhespirova, I.F., Kovalenko, Y., Álvarez, R., Prokhorov, E., and Ramos, A., 2011. Evolutionary testing of hydraulic simulator functionality. *Water resources management*, 25(8), pp.1935–1947. Available from: https://doi.org/10.1007/s11269-011-9782-5.

Jung, D. and Kim, J.H., 2018. Water distribution system design to minimize costs and maximize topological and hydraulic reliability. *Journal of water resources planning and management*, 144(9). Available from: https://doi.org/10.1061/(asce)wr.1943-5452.0000975.

Koritsas, E., Sidiropoulos, E., and Evangelides, C., 2018. Optimization of branched water distribution systems by means of a physarum—inspired algorithm. *Proceedings*, 2(11). Available from: https://doi.org/10.3390/proceedings2110598.

Kovalenko, Y., Gorev, N.B., Kodzhespirova, I.F., Prokhorov, E., and Trapaga, G., 2014. Convergence of a hydraulic solver with pressure-dependent demands. *Water resources management*, 28(4), pp.1013–1031. Available from: https://doi.org/10.1007/s11269-014-0531-4.

Lee, H.M., Yoo, D.G., Kim, J.H., and Kang, D., 2016. Hydraulic simulation techniques for water distribution networks to treat pressure deficient conditions. *Journal of water resources planning and management*, 142(4). Available from: https://doi.org/10.1061/(asce)wr.1943-5452.0000624.

Lin, M.-D., Liu, Y.-H., Liu, G.-F., and Chu, C.-W., 2007. Scatter search heuristic for least-cost design of water distribution networks. *Engineering optimization*, 39(7), pp.857–876. Available from: https://doi.org/10.1080/03052150701503611.

Liong, S.-Y. and Atiquzzaman, M., 2004. Optimal design of water distribution network using shu2ed complex evolution. *Journal of the institution of engineers*, 44 ().

Mahmoud, H.A., Savić, D., and Kapelan, Z., 2017. New pressure-driven approach for modeling water distribution networks. *Journal of water resources planning and management*, 143(8). Available from: https://doi.org/10.1061/(asce)wr.1943-5452.0000781.

Mala-Jetmarova, H., Sultanova, N., and Savic, D., 2018. Lost in optimisation of water distribution systems? a literature review of system design. *Water*, 10(3). Available from: https://doi.org/10.3390/w10030307.

Martin-Candilejo, A., Santillán, D., Iglesias, A., and Garrote, L., 2020. Optimization of the design of water distribution systems for variable pumping flow rates. *Water*, 12(2). Available from: https://doi.org/10.3390/w12020359.

Martínez-Bahena, B., Cruz-Chávez, M., Ávila-Melgar, E., Cruz-Rosales, M., and Rivera-Lopez, R., 2018. Using a genetic algorithm with a mathematical programming solver to optimize a real water distribution system. *Water*, 10(10). Available from: https://doi.org/10.3390/w10101318.

Mays, L., 1999. *Water distribution system handbook*. 1st ed. McGraw-Hill Education, p.912.

Mora-Melia, D., Iglesias-Rey, P.L., Martinez-Solano, F.J., and Fuertes-Miquel, V.S., 2013. Design of water distribution networks using a pseudo-genetic algorithm and sensitivity of genetic operators. *Water resources management*, 27(12), pp.4149–4162. Available from: https://doi.org/10.1007/s11269-013-0400-6.

Pacchin, E., Alvisi, S., and Franchini, M., 2016. Analysis of non-iterative methods and proposal of a new one for pressure-driven snapshot simulations with epanet. *Water resources management*, 31(1), pp.75–91. Available from: https://doi.org/10.1007/s11269-016-1511-7.

Paez, D., Suribabu, C.R., and Filion, Y., 2018. Method for extended period simulation of water distribution networks with pressure driven demands. *Water resources management*, 32(8), pp.2837–2846. Available from: https://doi.org/10.1007/s11269-018-1961-1.

Pietrucha-Urbanik, K. and Tchórzewska-Cieślak, B., 2018. Approaches to failure risk analysis of the water distribution network with regard to the safety of consumers. *Water*, 10(11). Available from: https://doi.org/10.3390/w10111679.

Qiu, M., Elhay, S., Simpson, A.R., and Alexander, B., 2019. Benchmarking study of water distribution system solution methods. *Journal of water resources planning and management*, 145(2). Available from: https://doi.org/10.1061/(asce)wr.1943-5452.0001030.

Reca, J., Martínez, J., Gil, C., and Baños, R., 2007. Application of several meta-heuristic techniques to the optimization of real looped water distribution networks. *Water resources management*, 22(10), pp.1367–1379. Available from: https://doi.org/10.1007/s11269-007-9230-8.

Reca, J., Martínez, J., and López, R., 2017. A hybrid water distribution networks design optimization method based on a search space reduction approach and a genetic algorithm. *Water*, 9(11). Available from: https://doi.org/10.3390/w9110845.

Rossman, L.A., n.d. *Epanet 2 users manual* (v.Version 2). English. (V.Version 2). U.S. Environmental Protection Agency. 200pp. September 01, 2000.

Sayyed, M.A.H.A., Gupta, R., and Tanyimboh, T.T., 2014. Modelling pressure deficient water distribution networks in epanet. *Procedia engineering*, 89, pp.626–631. Available from: https://doi.org/10.1016/j.proeng.2014.11.487.

Sebbagh, K., Safri, A., and Zabot, M., 2018. Pre-localization approach of leaks on a water distribution network by optimization of the hydraulic model using an evolutionary algorithm. *Proceedings*, 2(11). Available from: https://doi.org/10.3390/proceedings2110588.

Seyoum, A.G. and Tanyimboh, T.T., 2016. Investigation into the pressure-driven extension of the epanet hydraulic simulation model for water distribution systems. *Water resources management*, 30(14), pp.5351–5367. Available from: https://doi.org/10.1007/s11269-016-1492-6.

Seyoum, A.G. and Tanyimboh, T.T., 2017. Integration of hydraulic and water quality modelling in distribution networks: epanet-pmx. *Water resources management*, 31(14), pp.4485–4503. Available from: https://doi.org/10.1007/s11269-017-1760-0.

Shao, Y., Yao, H., Zhang, T., Chu, S., and Liu, X., 2019. An improved genetic algorithm for optimal layout of flow meters and valves in water network partitioning. *Water*, 11(5). Available from: https://doi.org/10.3390/w11051087.

Siew, C. and Tanyimboh, T.T., 2011. Pressure-dependent epanet extension: pressure-dependent demands. *Water distribution systems analysis 2010*, pp.75–84. Available from: https://doi.org/10.1061/41203(425)9.

Siew, C., Tanyimboh, T.T., and Seyoum, A.G., 2016. Penalty-free multi-objective evolutionary approach to optimization of anytown water distribution network. *Water resources management*, 30(11), pp.3671–3688. Available from: https://doi.org/10.1007/s11269-016-1371-1.

Skworcow, P., Paluszczyszyn, D., and Ulanicki, B., 2014. Pump schedules optimisation with pressure aspects in complex large-scale water distribution systems. *Drinking water engineering and science*, 7(1), pp.53–62. Available from: https://doi.org/10.5194/dwes-7-53-2014.

Tanyimboh, T.T., Siew, C., Saleh, S., and Czajkowska, A., 2016. Comparison of surrogate measures for the reliability and redundancy of water distribution systems. *Water resources management*, 30(10), pp.3535–3552. Available from: https://doi.org/10.1007/s11269-016-1369-8.

Trifunović, N., Abunada, M., Babel, M., and Kennedy, M., 2015. The role of balancing tanks in optimal design of water distribution networks. *Journal of water supply: research and technology-aqua*, 64(5), pp.610–628. Available from: https://doi.org/10.2166/aqua.2014.043.

Umapathi, S., Chong, M.N., and Sharma, A., 2012. Assessment of diurnal water demand patterns to determine supply reliability of plumbed rainwater tanks in south east queensland.

Vamvakeridou-Lyroudia, L.S., Savic, D.A., and Walters, G.A., 2007. Tank simulation for the optimization of water distribution networks. *Journal of hydraulic engineering*, 133(6), pp.625–636. Available from: https://doi.org/10.1061/(asce)0733-9429(2007)133:6(625).

Wang, Q., Wang, L., Huang, W., Wang, Z., Liu, S., and Savić, D.A., 2019. Parameterization of nsga-ii for the optimal design of water distribution systems. *Water*, 11(5). Available from: https://doi.org/10.3390/w11050971.

Wzorek, M., Boczar, T., Adamikiewicz, N., Stanisławski, W., Królczyk, G., and Król, A., 2017. Calibration of parameters of water supply network model using genetic algorithm. *E3s web of conferences*, 19. Available from: https://doi.org/10.1051/e3sconf/20171902007.

Zhang, H., Cheng, X., Huang, T., Cong, H., and Xu, J., 2017. Hydraulic analysis of water distribution systems based on fixed point iteration method. *Water resources management*, 31(5), pp.1605–1618. Available from: https://doi.org/10.1007/s11269-017-1601-1.

A. Source code for pipeline diameter optimisation program

The Python source code for the genetic algorithm pipeline diameter optimisation program is appended in the following pages.

The source code and packaged executable are available on this program's GitHub hosting page at github.com/bowenfan96/epanet-genetic-algorithm. If updates or improvements are made to this program, it will be reflected on the GitHub hosting page along with its version history.

Any feedback is welcome. They can be raised on the GitHub Issues page.

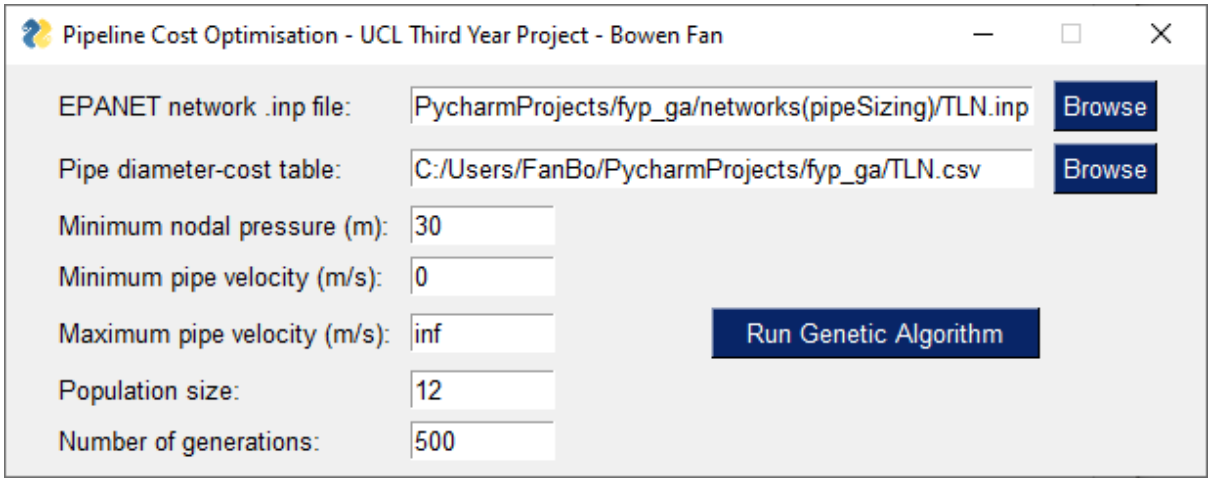


Figure A.1.: Genetic algorithm program interface

```python
# Genetic Algorithm for the optimisation of pipeline diameters
# Written by Bowen Fan, UCL

import epamodule as ep
import numpy as np
import PySimpleGUI as sg
import matplotlib.pyplot as plt
import csv
import random

# global variables
commercial_diameters = []
commercial_costs = []
ridiculous_value = float('inf')


def pipe_cost(diameter):
    cost_index = commercial_diameters.index(diameter)
    pipe_cost = commercial_costs[cost_index]
    return pipe_cost


def network_cost(length_array, diameter_array):
    assert len(length_array) == len(diameter_array)
    network_cost = 0
    for i in range(len(length_array)):
        network_cost += length_array[i] * (pipe_cost(diameter_array[i]))
    return network_cost


def constraint_checker(diameter_array, min_pre, min_vel, max_vel,
                       file, num_nodes, num_links, num_pipes):
    # reconstruct .inp file from diameter array and store it as a temp file
    for pipe in range(len(diameter_array)):
        if diameter_array[pipe] > 0:
            ep.ENsetlinkvalue(pipe + 1, 0, diameter_array[pipe])

    ep.ENsaveinpfile(file)

    ep.ENopenH()
    ep.ENinitH()
    ep.ENrunH()

    # initialize an array of nodal heads
    head_array = np.zeros(num_nodes)
    # initialize an array of pipe velocities
    velocity_array = np.zeros(num_links)

    fail_pressure_list = []
    fail_min_vel_list = []
    fail_max_vel_list = []
    negative_diameter_list = []

    fail_pressure = False
    fail_min_vel = False
    fail_max_vel = False
    negative_diameter = False

    for node in range(num_nodes):

        if ep.ENgetnodevalue(node + 1, 11) < min_pre and ep.ENgetnodetype(node + 1) ==
0:
            fail_pressure_list.append(node)
            fail_pressure = True

    for pipe in range(num_pipes):
```

```
66
67          if ep.ENgetlinkvalue(pipe + 1, 9) < min_vel:
68              fail_min_vel_list.append(pipe)
69              fail_min_vel = True
70          elif ep.ENgetlinkvalue(pipe + 1, 9) > max_vel:
71              fail_max_vel_list.append(pipe)
72              fail_max_vel = True
73
74      i = 0
75      for diameter in diameter_array:
76          i += 1
77          if diameter <= 0:
78              negative_diameter_list.append(i)
79              negative_diameter = True
80
81      print('Fail pressure list (nodes):', fail_pressure_list)
82      print('Lower than min velocity list (links):', fail_min_vel_list)
83      print('Higher than max velocity list (links):', fail_max_vel_list)
84
85      if fail_pressure or fail_min_vel or fail_max_vel or negative_diameter:
86          return False
87      else:
88          return True
89
90
91  def random_diameter_generator(num_pipes):
92      random_diameter_array = []
93      for pipe in range(num_pipes):
94          random_diameter_array.append(random.choice(commercial_diameters))
95      assert len(random_diameter_array) == num_pipes
96      return random_diameter_array
97
98
99  def fitness_function(cost, constraint_pass):
100     if constraint_pass:
101         return cost
102     else:
103         return ridiculous_value
104
105
106 def population_cost(length_array, population_array, population_size, min_pre, min_vel
    , max_vel,
107                     file, num_nodes, num_links, num_pipes):
108     # this function calculates the cost of each diameter configuration in the
    population
109     # and append it to an array listing the fitness values
110     # if the network configuration fails the pressure or velocity constraints, then it
     receives an infinite penalty
111     # i.e. the cost of infeasible networks is set at infinity, so they will not be
    chosen for the next generation
112
113     config_costs = np.zeros(population_size)
114
115     for diameter_config in range(population_size):
116         current_config = population_array[diameter_config, :]
117         config_costs[diameter_config] = fitness_function(network_cost(length_array,
    current_config),
118                                                          constraint_checker(
    current_config, min_pre, min_vel,
119                                                                             max_vel,
    file, num_nodes,
120                                                                             num_links,
    num_pipes))
121     return config_costs
122
123
```

```
124 def get_best_configs(population, config_fitness, num_parents):
125     best_configs = np.zeros((num_parents, population.shape[1]))
126 
127     for config in range(num_parents):
128         best_config_index = np.where(config_fitness == np.min(config_fitness))[0][0]
129         best_configs[config, :] = population[best_config_index, :]
130         config_fitness[best_config_index] = ridiculous_value
131 
132     return best_configs
133 
134 
135 def merge_best_configs(best_configs, num_children, num_pipes):
136     evolved_generation = np.zeros((num_children, num_pipes))
137 
138     merge_index = np.random.randint(0, num_pipes)
139 
140     # merge the best configurations into evolved configurations in a rotary relay
141     # i.e. 1 merge with 2, 2 with 3, 3 with 4, 4 with 1
142 
143     for i in range(num_children):
144         if i < best_configs.shape[0]:
145 
146             config_1 = i % best_configs.shape[0]
147             config_2 = (i + 1) % best_configs.shape[0]
148 
149             evolved_generation[i, 0:merge_index] = best_configs[config_1, 0:
    merge_index]
150             evolved_generation[i, merge_index:] = best_configs[config_2, merge_index:]
151 
152         else:
153             evolved_generation[i, :] = random_diameter_generator(num_pipes)
154 
155     return evolved_generation
156 
157 
158 def mutate_config(evolved_generation, mutations=1):
159     for pipe_config in range(evolved_generation.shape[0]):
160         for mutation in range(mutations):
161             evolved_generation[pipe_config, np.random.randint(0, evolved_generation.
    shape[1])] \
162                 = random.choice(commercial_diameters)
163 
164     return evolved_generation
165 
166 
167 def main():
168     inp_file = values['inp_filepath']
169     csv_file = values['csv_filepath']
170     min_pre = float(values['min_pressure'])
171     min_vel = float(values['min_velocity'])
172     max_vel = float(values['max_velocity'])
173     population_size = int(values['pop_size'])
174     num_generations = int(values['num_generations'])
175 
176     with open(csv_file, 'r') as cost_file:
177         read_csv = csv.reader(cost_file, delimiter=',')
178         next(read_csv)
179         for rows in read_csv:
180             commercial_diameters.append(float(rows[0]))
181             commercial_costs.append(float(rows[1]))
182 
183     assert len(commercial_diameters) == len(commercial_costs)
184 
185     # call the EPANET engine to open the supplied network input file
186     ep.ENopen(inp_file)
187 
```

```
188     # get from EPANET the number of nodes in the network (nodes: junctions, reservoirs
    , tanks)
189     num_nodes = ep.ENgetcount(0)
190 
191     # get from EPANET the number of tanks and reservoirs in the network
192     num_tanks = ep.ENgetcount(1)
193 
194     # get from EPANET the number of links in the network (links: pipes, valves, pumps)
195     num_links = ep.ENgetcount(2)
196 
197     # get from EPANET the number of pipes in the network
198     num_pipes = 0
199     for link in range(num_links):
200         if ep.ENgetlinktype(link + 1) == 0 or 1:
201             num_pipes += 1
202 
203     # initialize an array of pipe diameters
204     diameter_array = np.zeros(num_pipes)
205     # initialize an array of pipe lengths
206     length_array = np.zeros(num_pipes)
207 
208     # this FOR loop makes 2 tables, listing the lengths and diameters of the pipes
209     for pipe in range(num_pipes):
210         diameter_array[pipe] = ep.ENgetlinkvalue(pipe + 1, 0)
211         length_array[pipe] = ep.ENgetlinkvalue(pipe + 1, 1)
212 
213     # check that the diameter and length arrays contain the same number of pipes
214     assert len(diameter_array) == len(length_array)
215 
216     # initialize a randomly generated population array
217     # the population size is the number of pipe configurations
218     population_array = []
219     for network in range(population_size):
220         population_array.append(random_diameter_generator(num_pipes))
221     population_array = np.array(population_array)
222 
223     least_cost_configs = []
224     num_configs_to_merge = 4
225 
226     # genetic algorithm iteration loop
227     for ga_generation in range(num_generations):
228 
229         # call the EPANET engine to open the file in each loop
230         ep.ENopen(inp_file)
231 
232         print('Current Generation: ', ga_generation)
233 
234         # get the cost of each pipe network configuration in the generation using the
    cost functions above
235         # the cost is listed in an array corresponding to the cost of each network
    configuration
236         config_costs = population_cost(length_array, population_array, population_size
    , min_pre, min_vel, max_vel,
237                                        inp_file, num_nodes, num_links, num_pipes)
238         print('Network Costs: ')
239         print(config_costs)
240 
241         # record the cost of the lowest cost feasible configuration found so far
242         least_cost_configs.append(np.min(config_costs))
243         print('Lowest Cost: ', np.min(config_costs))
244 
245         # get the lowest cost configurations to merge (mating function)
246         # these are the least cost parents that we will merge to get evolved children
247         # (which are hopefully better than both the parents)
248         best_configs = get_best_configs(population_array, config_costs,
    num_configs_to_merge)
```

```
249         print('Parent Configurations: ')
250         print(best_configs)
251 
252         # evolve by merging the best configurations (crossover function)
253         num_children = population_size - num_configs_to_merge
254         evolved_children = merge_best_configs(best_configs, num_children, num_pipes)
255         print('Evolved Children Configurations: ')
256         print(evolved_children)
257 
258         # mutate some pipes in the children for genetic diversity (mutation function)
259         mutated_evolved_children = mutate_config(evolved_children, mutations=2)
260         print('Evolved and Mutated Configurations: ')
261         print(mutated_evolved_children)
262 
263         # append the evolved and mutated children to the parents
264         # e.g. if the population size is 10, and the top 4 least cost configs are
    chosen as parents, then:
265         # the 4 least cost parents will merge to output 4 evolved children configs:
266         # 1 will merge with 2, 2 with 3, 3 with 4, 4 with 1
267 
268         # another 2 network configs will be randomly generated from the commercially
    available diameters
269         # to add genetic diversity
270 
271         # thus, we now have 4 least cost parents, 4 evolved children (which are
    hopefully
272         # even lower cost than their parents), and 2 randomly generated configs
273         # these are first mutated, then merged to get back the population size of 10 (
    4 + 4 + 2)
274 
275         population_array[0: num_configs_to_merge, :] = best_configs
276         population_array[num_configs_to_merge:, :] = mutated_evolved_children
277 
278         # VERSION 18: if best cost is inf (means every network is infeasible), then
    increase all pipe by 1 size
279         # the old genetic algorithm in ga version 17 got stumped by the Hanoi network
    and fails to converge
280         # this is because all the configs were inf - hence no comparison and selection
     and improvement could be made
281         if np.min(least_cost_configs) == float('inf'):
282             with np.nditer(population_array, op_flags=['readwrite']) as iterator:
283                 for pipe in iterator:
284                     # print("Pipe size: ", pipe)
285                     pipe_index = np.where(commercial_diameters == pipe)[0][0]
286                     # print("Pipe index: ", pipe_index)
287                     if pipe < np.max(commercial_diameters):
288                         pipe[...] = commercial_diameters[pipe_index + 1]
289                         # print("Size increased to: ", pipe)
290 
291         print('New Population: ')
292         print(population_array)
293 
294         # close the EPANET engine to free up memory for the next generation
295         ep.ENclose()
296 
297         # end of the generation algorithm iteration loop
298 
299     # this segment writes the least cost config into the EPANET input file and prints
    the config and its cost
300     ep.ENopen(inp_file)
301 
302     # get the lowest cost feasible network config found by the genetic algorithm
303     final_costs = population_cost(length_array, population_array, population_size,
    min_pre, min_vel, max_vel,
304                                   inp_file, num_nodes, num_links, num_pipes)
305 
```

```
306     least_cost_index = np.where(final_costs == np.min(final_costs))[0][0]
307     least_cost_network = population_array[least_cost_index, :].flatten()
308 
309     # write the least cost feasible network config back into the supplied EPANET input
     file
310     for pipe in range(len(least_cost_network)):
311         ep.ENsetlinkvalue(pipe + 1, 0, least_cost_network[pipe])
312 
313     ep.ENsaveinpfile(inp_file)
314     ep.ENclose()
315 
316     # print the least cost network and its cost to the console
317     print('Least Cost Network Found by Genetic Algorithm: ')
318     print(least_cost_network)
319     print('Cost of Least Cost Network: ')
320     print(final_costs[least_cost_index])
321 
322     if final_costs[least_cost_index] == float('inf'):
323         print('No solutions found. Problem may be overconstrained or insufficient
    number of generations.')
324         sg.Popup('No solutions found. Problem may be overconstrained or insufficient
    number of generations.')
325 
326     # use matplotlib to plot the best cost against number of generations
327     plt.plot(least_cost_configs)
328     plt.xlabel("Generation")
329     plt.ylabel("Cost of the Least Cost Network")
330     plt.show()
331 
332 
333 # GUI window using PySimpleGUI
334 
335 layout = [[sg.Text('EPANET network .inp file:', size=(21, 1)),
336            sg.Input(key='inp_filepath',
337                     default_text='C:/Users/FanBo/PycharmProjects/fyp_ga/networks(
    pipeSizing)/BIN/BIN.inp'),
338            sg.FileBrowse(file_types=(("INP", ".inp"),))],
339 
340           [sg.Text('Pipe diameter-cost table:', size=(21, 1)),
341            sg.Input(key='csv_filepath',
342                     default_text='C:/Users/FanBo/PycharmProjects/fyp_ga/BIN.csv'),
343            sg.FileBrowse(file_types=(("CSV", ".csv"),))],
344 
345           [sg.Text('Minimum nodal pressure (m):', size=(21, 1)),
346            sg.Input(key='min_pressure', size=(10, 1),
347                     default_text=30)],
348 
349           [sg.Text('Minimum pipe velocity (m/s):', size=(21, 1)),
350            sg.Input(key='min_velocity', size=(10, 1),
351                     default_text=0)],
352 
353           [sg.Text('Maximum pipe velocity (m/s):', size=(21, 1)),
354            sg.Input(key='max_velocity', size=(10, 1),
355                     default_text='inf'),
356 
357            sg.Text('', size=(7, 1)),
358            sg.Button('Run Genetic Algorithm', size=(20, 1))],
359 
360           [sg.Text('Population size:', size=(21, 1)),
361            sg.Input(key='pop_size', size=(10, 1),
362                     default_text=12)],
363 
364           [sg.Text('Number of generations:', size=(21, 1)),
365            sg.Input(key='num_generations', size=(10, 1),
366                     default_text=500)],
367           ]
```

```
368
369 window = sg.Window('Pipeline Cost Optimisation - UCL Third Year Project - Bowen Fan',
    layout)
370
371 while True:
372     action, values = window.Read()
373
374     if action == 'Run Genetic Algorithm':
375         main()
376
377     elif action is None:
378         break
379
380     break
381
382 window.Close()
383
```

B. Source code for buffer tank location optimisation program

The Python source code for the buffer tank location optimisation program is appended in the following pages.

The source code and packaged executable are available on this program's GitHub hosting page at github.com/bowenfan96/demand-balancing-tanks. If updates or improvements are made to this program, it will be reflected on the GitHub hosting page along with its version history.

Any feedback is welcome. They can be raised on the GitHub Issues page.

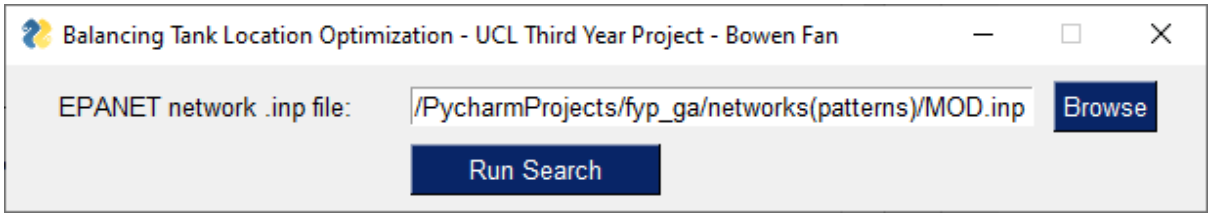


Figure B.1.: Buffer tank optimisation program interface

```
# Brute-force search algorithm for the optimisation of buffer tank nodal location
# Written by Bowen Fan, UCL

from epynet import Network
import numpy as np
import PySimpleGUI as sg
import pandas as pd

score_index_counter = 0
tank_elev_array = []


def solve_and_return_pressures(num_junctions):
    # function to calculate the pressure of every node at any timestep

    # call EPANET to load the network

    network = Network(inp_file)
    network.ep.ENopen(inp_file)

    network.ep.ENopenH()
    network.ep.ENinitH()

    # temp array for storing pressure of each junction at each timestep
    temp_pressure_array = np.array([])

    # big array of all pressures for all junctions at all timesteps
    perm_pressure_array = np.empty((num_junctions, 0))

    # solve for all timesteps and store each node's pressure at each timestep in an array
    runtime = 0

    while network.ep.ENnextH() > 0:

        network.ep.ENrunH()

        for junction in range(num_junctions):
            junction_pressure = network.ep.ENgetnodevalue(junction + 1, 11)
            temp_pressure_array = np.append(temp_pressure_array, junction_pressure)

        perm_pressure_array = np.append(perm_pressure_array, temp_pressure_array.reshape(num_junctions, 1), axis=1)

        runtime += network.ep.ENgettimeparam(1)
        temp_pressure_array = []

    network.ep.ENcloseH()
    network.ep.ENdeleteproject()

    return perm_pressure_array


def score_pressure_array(perm_pressure_array):
    global score_index_counter

    # array of pressures averaged across all junctions, for each timestep
    avg_pressure_array = np.array([])

    # minimum network pressure, for each timestep
    min_pressure_array = np.array([])

    # minimum average network pressure, this should occur at the time of peak demand. THIS IS THE SCORE RETURNED.
    min_average_pressure = np.array([])

```

```
64      # average pressure at each junction, averaged across all timesteps
65      avg_pressure_at_each_junc = np.array([])
66
67      avg_pressure_array = np.mean(perm_pressure_array, axis=0)
68      min_pressure_array = np.min(perm_pressure_array, axis=0)
69      min_average_pressure = np.min(avg_pressure_array)
70
71      score_index_counter += 1
72
73      return min_average_pressure
74
75
76  def average_initial_pressures(perm_pressure_array):
77      # simple function to average all nodal pressures in the network
78      avg_pressure_at_each_junc = np.mean(perm_pressure_array, axis=1)
79
80      return avg_pressure_at_each_junc
81
82
83  def add_tank_get_score(num_junctions, avg_initial_pressure_at_each_junc):
84      global tank_elev_array
85      network = Network(inp_file)
86
87      junc_elev_array = []
88      junc_x_y_array = []
89      junc_id_array = []
90
91      score_array = []
92
93      # this loop add a tank to Node 1, calculate the peak-demand average pressure,
    stores it, and deletes the tank
94      # it then adds a tank to Node 2 and repeats until all nodal locations are scored
95
96      for junction in range(num_junctions):
97          junc_elev_array.append(network.ep.ENgetnodevalue(junction + 1, 0))
98          junc_x_y_array.append(network.ep.ENgetcoord(junction + 1))
99          junc_id_array.append(network.ep.ENgetnodeid(junction + 1))
100
101         optimum_elevation = junc_elev_array[junction] +
    avg_initial_pressure_at_each_junc[junction]
102
103         tank_elev_array.append(optimum_elevation)
104
105         network.add_tank(uid='balancing_tank', x=junc_x_y_array[junction][0], y=
    junc_x_y_array[junction][1],
106                          elevation=optimum_elevation, diameter=100, maxlevel=1000,
    minlevel=0, tanklevel=0)
107
108         network.add_pipe(uid='balancing_tank_pipe', from_node=junc_id_array[junction],
109                          to_node='balancing_tank', diameter=1000, length=10, roughness
    =1E9, check_valve=False)
110
111         network.save_inputfile(inp_file)
112
113         junction_score = score_pressure_array(solve_and_return_pressures(num_junctions
    ))
114
115         score_array.append(junction_score)
116         print("Junction index ", score_index_counter, " : ", junction_score)
117
118         network.delete_link('balancing_tank_pipe')
119         network.delete_node('balancing_tank')
120         network.reset()
121
122         network.save_inputfile(inp_file)
123
```

```
124     return score_array
125 
126 
127 def main():
128     network = Network(inp_file)
129 
130     num_junctions = 0
131     num_nodes = network.ep.ENgetcount(0)
132 
133     for node in range(num_nodes):
134         if network.ep.ENgetnodetype(node + 1) == 0:
135             num_junctions += 1
136 
137     print("Number of junctions: ", num_junctions, "\n")
138 
139     avg_initial_pressure_at_each_junc = average_initial_pressures(
    solve_and_return_pressures(num_junctions))
140 
141     scores = add_tank_get_score(num_junctions, avg_initial_pressure_at_each_junc)
142 
143     best_junction = network.ep.ENgetnodeid(int(np.argmax(scores) + 1))
144 
145     junction_ids = []
146 
147     for junction in range(num_junctions):
148         junction_ids.append(network.ep.ENgetnodeid(junction + 1))
149 
150     print("\nIndex of best junction: ", np.argmax(scores) + 1)
151 
152     print("ID of best junction: ", best_junction, "\nMin avg network pressure with
    optimized tank: ", np.max(scores))
153 
154     original_score = score_pressure_array(solve_and_return_pressures(num_junctions))
155 
156     print("\nMin avg network pressure without buffer tank: ", original_score)
157 
158     score_dataframe = pd.DataFrame({"Junction ID":junction_ids,
159                                     "Min Avg Network Pressure":scores,
160                                     "Tank Elevation":tank_elev_array})
161 
162     score_dataframe.index = score_dataframe.index + 1
163     score_dataframe.index.name = "Junction Index"
164 
165     sorted_scores = score_dataframe.sort_values("Min Avg Network Pressure", ascending=
    False)
166 
167     print("\nTank locations sorted by minimum average network pressure:\n")
168     print(sorted_scores)
169 
170     print("\nSaving CSV...")
171     sorted_scores.to_csv("Optimal_Location_List.csv")
172 
173 
174 # GUI window using PySimpleGUI
175 
176 layout = [[sg.Text('EPANET network .inp file:', size=(21, 1)),
177            sg.Input(key='inp_filepath',
178                     default_text='C:\\Users\\FanBo\\PycharmProjects\\fyp_ga\\pattern2.
    inp'),
179            sg.FileBrowse(file_types=(("INP", ".inp"),))],
180 
181           [sg.Text('', size=(21, 1)),
182            sg.Button('Run Search', size=(15, 1))]
183           ]
184 
185 window = sg.Window('Balancing Tank Location Optimization - UCL Third Year Project -
```

```
185 Bowen Fan', layout)
186
187 while True:
188     action, values = window.Read()
189
190     inp_file = values['inp_filepath']
191
192     if action == 'Run Search':
193         main()
194
195     elif action is None:
196         break
197
198     break
199
200 window.Close()
201
```