UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

# TECHNO-OPTIMIZATION OF $CO_2$ TRANSPORT NETWORKS WITH CONSTRAINED PIPELINE PARAMETERS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

DAVID NNAMDI NNAMDI

Norman, Oklahoma

2023

TECHNO-OPTIMIZATION OF $CO_2$ TRANSPORT NETWORKS WITH CONSTRAINED PIPELINE PARAMETERS

A THESIS APPROVED FOR THE
MEWBOURNE SCHOOL OF PETROLEUM AND GEOLOGICAL ENGINEERING

BY THE COMMITTEE CONSISTING OF

Dr. Rouzbeh G. Moghanloo, Committee Chair

Dr. Deepak Devegowda, Committee Member

Dr. Sean Yaw, Committee Member

Dedicated to God.

## ACKNOWLEDGEMENTS

I would like to thank my research advisor, Dr. Rouzbeh Moghanloo, for his tremendous support and guidance throughout my Master's Degree. Under his tutelage, I have gained extensive exposure and research experience in Carbon Sequestration and economic optimization through several projects over the years. Special thanks to all the members of the Carbon Utilization and Storage Partnership (CUSP). To Dr. Sean Yaw and DaneshFar Jamal, I appreciate the time I spent working with you and for all the creative guidance in my earlier publications.

I would also like to thank all the professors I worked under as a Teaching Assistant during my time at OU, to Dr. Ali Tinni, Dr. Mohammed, Late Dr. Carl Sondergeld and finally Dr. Rai. I can proudly say I worked for some of the best professors the department had to offer, and I am very grateful for that opportunity. Special thanks are in order to Katie Shapiro, Sonya Grant and Francey Freeman for always responding to my many requests for information and academic related help. Thank you to my OU SPE family where I humbly served as external liaison, which was one of the most wonderful experiences I had at OU.

Finally, I would like to thank my parents: Mr. and Mrs. Nnamdi Nosike for all their prayers and encouragements, and my siblings: Jessie, Melissa, Nancy for always offering me words of comfort when I felt down and alone. To my closest friends, Ayomide Hamzat, Karen Ochie and Chinedu Nwosu, thank you for being my source of encouragement every day, this journey is very special because of you all. All the other members of the Nigerian OU community, my wonderful friends, thank you for genuine friendship, I will forever cherish all the memories we made.

# Table of Contents

## LIST OF FIGURES

# LIST OF TABLES

# ABSTRACT

In planning large scale carbon sequestration projects, one of the key parameters affecting project economics is the selection of optimal pipeline transportation networks connecting physical locations of carbon sources to sinks (or injection sites). This network is usually determined based on several limiting factors including existing right-of-way, densely populated regions, topology, etc. Open-source tools such as SimCCS$^{2.0}$ do an effective job in proposing provably optimal routes for construction of new pipelines but are unable to accommodate existing pipelines in techno-economic optimization. With the newly amended 45Q laws offering 70% more tax credits for carbon sequestration than it did in the 2018 amendment, energy companies are looking more into repurposing gas and liquid transportation lines for $CO_2$ transportation to abandoned oil and gas wells for carbon storage and this has further bolstered the need to have a method to account for existing pipelines in sequestration economics.

This project demonstrates a method to account for existing pipelines by 1 introducing zero cost paths into the cost surface to represent pipelines, 2 allowing for tie points into the existing pipeline by use of cost exclusion zones around zero cost paths and then, 3 calculating least cost paths and defining transshipment nodes along pipeline intersections. Doing this allowed for a reformulation of the alternate network paths between sources and sinks, and the network was then solved as Minimum-Cost-Network-Flow-Problem (MCNFP) modeled as a mixed integer programming problem.

The solution was developed using Python programming language and demo test cases are shown to illustrate the effectiveness of the solution in assessing cost reduction associated with $CO_2$ transfer from sources tied into locations along existing transport pipelines to sinks.

This solution has been packaged into a software name Sequestrix and has been made publicly available on GitHub for researchers and economic analysts to take advantage of for evaluating large scale CCUS projects, and to encourage further development and collaboration.

# CHAPTER 1: Introduction

The concentration of $CO_2$ in the atmosphere has rapidly increased since the start of the industrial revolution in 1750, this increase is primarily due to $CO_2$ emissions rates being increasingly higher than the rates at which natural sinks on land (via as plants and micro-organisms) and oceans (via inorganic dissolution) can absorb $CO_2$ (NOAA, 2022). This occurrence, as illustrated in figure 1, has led to increasing effects of global warming which have become impossible to dismiss as any form of scientific conspiracy to promote an increasing share of renewable energy sources in the global energy mix.

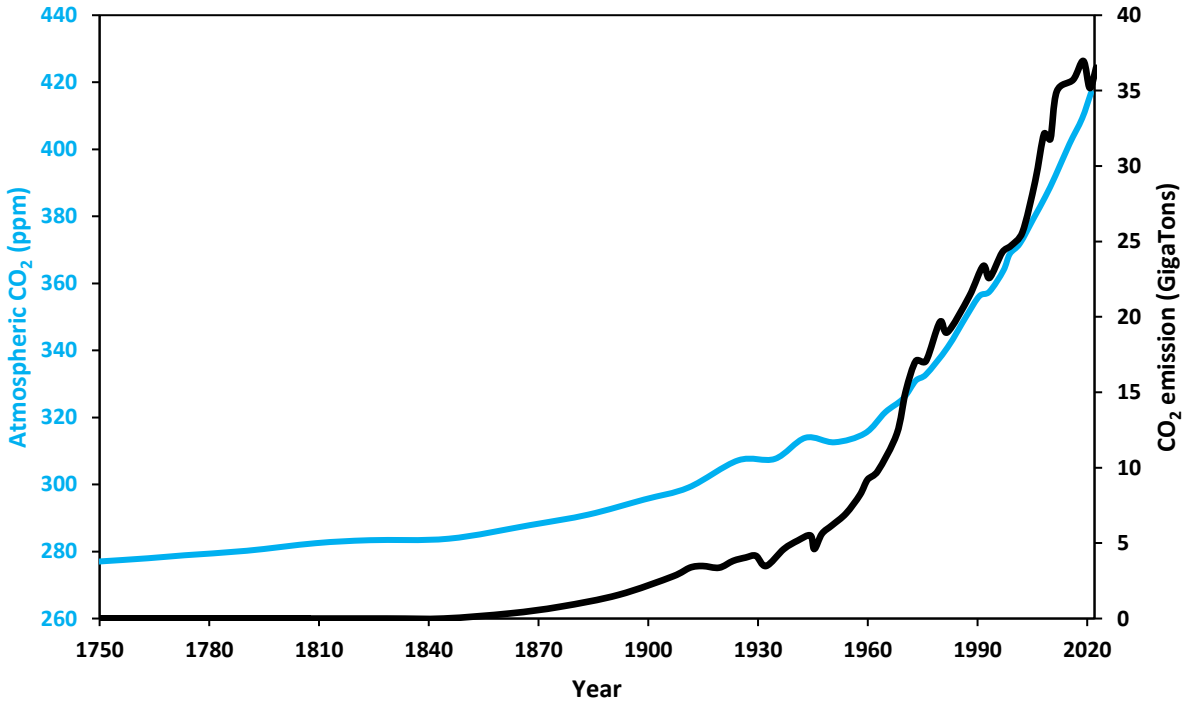



*Figure 1: Changes in atmospheric $CO_2$ concentration (blue line) with $CO_2$ emissions (black line) since the start of the industrial revolution (1750) till late 2021. Source (NOAA climate.gov)*

According to the Intergovernmental Panel on Climate Change, IPCC, the effects of climate change can be (and is projected to be) felt across multiple spheres including (1) ecosystem disruptions – where increasing temperatures in the arctic sea has led to increased migration of species from warmer land and sea areas and declining population of ice-dependent species such as polar bears, (2) agriculture and food security – where rapid changes in temperature is causing disruptions to harvest stability and livestock yield, (3) to human communities, livelihoods and lifestyles – where climate change has led to increased wildfires close to large settlements (e.g., in California, USA), increased risk of flooding and displacement in coastal settlements where majority of the residents are low – middle income earners (IPCC, 2022),. Figure 2 below illustrates the interactions between these spheres and the associated risk levels.

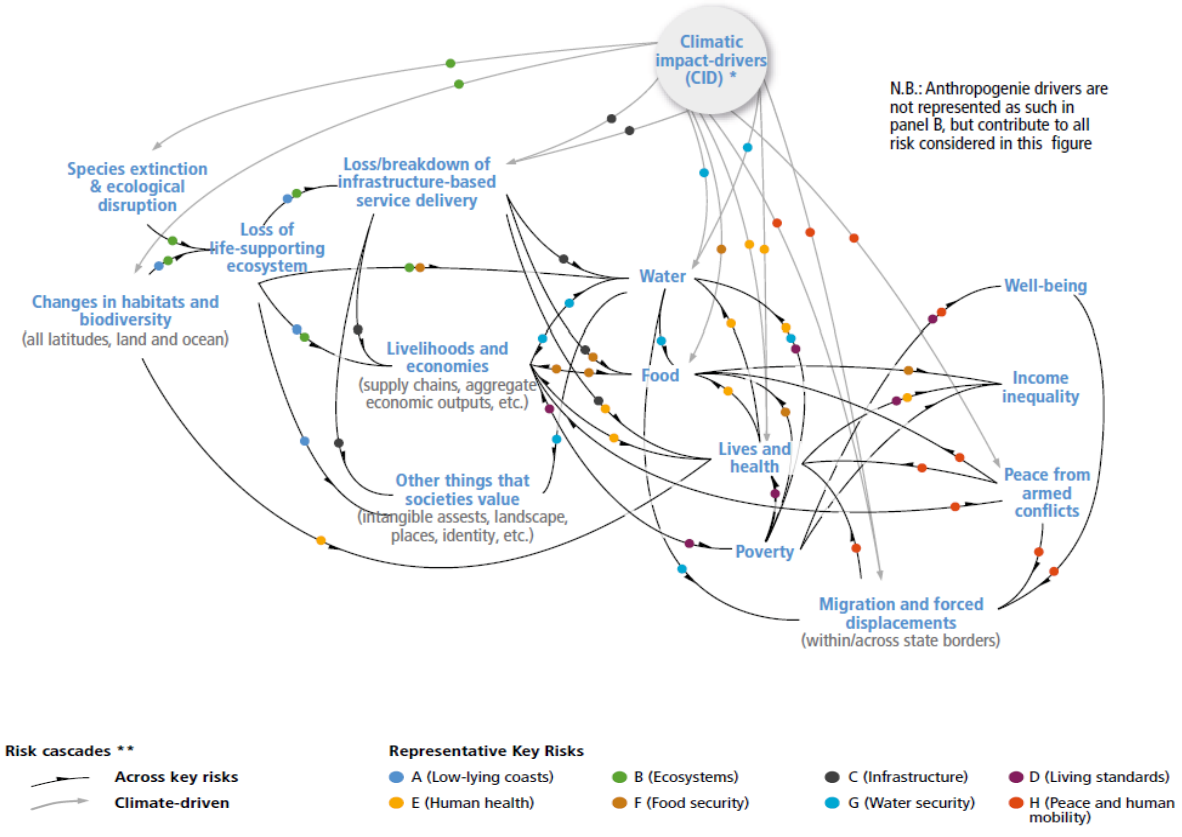



*Figure 2: Illustration of interactions across key risks associated with climate change (IPCC, 2022)*

The primary source of climate change has been linked to greenhouse gas emissions, particularly carbon dioxide and methane. The Petroleum industry has for decades been a major contributor to these emissions which is sustained by a growing demand for energy as 3rd world populations move from poverty into middle class economic status.

The current world energy outlook (IEA, 2022) suggests that fossil fuels will still play a role in energy generation up to 2035 and beyond due to the growing energy demands with renewable energy production growing significantly with advancements in technology. World governments and climate researchers had realized this fact a few years ago and started to proffer solutions to greenhouse gas emissions. One of the major solutions proffered was carbon capture and sequestration.

Carbon capture and sequestration involves capturing $CO_2$ directly from emission sources or from the atmosphere via direct air capture and injecting (or storing) it in geological formations. In the oil and gas industry, the practice of $CO_2$ injection started in the 1970s as a means for improving oil production from reservoirs (Núñez-López & Moskal, 2019), this process is called $CO_2$ EOR. Oil and $CO_2$ are miscible at defined reservoir pressure and temperature conditions and the dissolved $CO_2$ reduces the oil density and alters interfacial tension thus improving mobility making it easier for the oil to travel within pore spaces.

In more recent times, the concept of storing $CO_2$ in deep saline aquifers have become more mainstream as they offer vast potential storage for captured $CO_2$ (McPherson & Cole, 2000). One of the major challenges involved in developing large scale $CO_2$ sequestration projects, especially in saline aquifers, is project economics. In contrast to $CO_2$ EOR where capture and storage costs

may be offset by additional oil production, sequestration in saline aquifers offer no direct economic reward.

Out of the three major costs for $CO_2$ sequestration, injection costs are known with high certainty based on decades of oil and gas operations, capture costs are site specific and based on current technology can also be reasonably estimated to range between 11 and 75 $/metric ton of $CO_2$ (Abramson et al., 2020), however, transportation costs require a level of sophistication to adequately estimate. $CO_2$ can be transported via several means including – rail cargos, ships, trucks, and pipelines, however, most large-scale $CO_2$ projects require pipeline transportation connecting a network of sources and sinks across a large geographic area.

Before these pipelines are constructed and during the project scoping phase, a method for estimating the best and most cost-effective pipeline route must be established and this is usually achieved via graph network optimization. There has been recent research and advancement in the field, with working solutions such as SimCCS (Middleton & Bielicki, 2009) and SimCCS$^{2.0}$ (Middleton et al., 2020) developed and currently commercialized under Carbon Solutions LLC. These tools can calculate provably correct pipeline routes between multiple geo locations and estimate resultant transport costs for given transport volume of $CO_2$. While these tools can suggest/calculate routes for new pipeline construction for use in economical evaluation of sequestration projects, they have one major limitation – handling existing pipelines.

With the recent drive by energy companies to meet environmental, social and government (ESG) goals for $CO_2$ reduction, and due to the recent increase in 45Q tax credits (26 U.S.C. § 45Q), operators are seeking to repurpose existing pipelines to transport $CO_2$ from sources to sequestration

sites and require a method for accounting for existing pipeline routes, capacities, and cost in economic evaluation of sequestration projects.

## 1.1 Scope of Thesis

This thesis introduces a method to bridge the gap in the current computational framework needed for assessing sequestration economics, taking into account preferred existing pipeline routes and constraints. It expands on existing knowledge of creating potential routes for new pipelines, allowing the integration of existing pipelines and adjustment of flow constraints before solving the $CO_2$ network optimization problem. The thesis offers a recommended workflow for manual entry of tie-in points or mathematical estimation, which operators can apply in sequestration projects.

## 1.2 Working Hypothesis

The working hypothesis for this thesis is that existing pipelines can be embedded by modifying the cost surface graph with zero cost paths thereby forcing shortest path algorithms to divert flow via existing pipelines when it is cost effective.

## 1.3 Organization of Thesis

This thesis is divided into five chapters which are cover the following themes:

- Chapter 1: Introduces background and motivation of this study, describes the scope, working hypothesis and organization of the thesis.
- Chapter 2: Literature review on $CO_2$ sequestration history, current sequestration infrastructure, economics, and network optimization

- Chapter 3: Describes the methodology for translating pipeline coordinates to graph coordinates, modifying cost surface graph, assigning or calculating tie-in points and generating alternate pipeline networks for sequestration. Network optimization and visualization frameworks utilized are also discussed.
- Chapter 4: Introduces new tool called Sequestrix and highlights the results of 3 demo cases that benchmarks new tool performance against SimCCS and showcases pipeline embedding capabilities for $CO_2$ sequestration projects.
- Chapter 5: Summarizes key conclusions and outlines future work.

# CHAPTER 2: Literature Review

## 2.1 $CO_2$ Sequestration History

$CO_2$ Sequestration has a long-related history with Enhanced Oil Recovery (EOR) projects, which sought to improve oil production from declining reservoirs. Although the first carbon capture plant was proposed in the late 1930's, the first large scale $CO_2$ underground injection project began in Sharon Ridge oilfield, Texas in 1972 (Núñez-López & Moskal, 2019) where ExxonMobil utilized the $CO_2$ for EOR. The Sleipner project, which was launched in 1996, is the world's first true integrated Carbon Capture and Storage (CCS) project and it was developed by Statoil to avoid carbon taxes imposed by the Norwegian government (Beckwith, 2011). According to the Global CCS Institute, as of 2022, there are 30 commercially operating CCS projects worldwide with 11 more being constructed and over 100 in different development stages.

Defined as the injection and permanent storage of carbon dioxide in underground geologic formations, $CO_2$ sequestration requires a closed loop path for some or all of the carbon injected such that there is minimal recycling back into the atmosphere (Hepburn et al., 2019). This definition has some implications for how Carbon sequestration is assessed in EOR and via injection in saline aquifers. $CO_2$ storage is also possible in Coal Beds to recover methane, (Gorucu et al., 2005) and Shales but with greater difficulty due to very low permeabilities, however the $CO_2$ trapping mechanism will be by adsorption (Fakher & Imqam, 2019).

### 2.1.1 Sequestration in Hydrocarbon Reservoirs

In EOR operations, the $CO_2$ which is injected with water in alternating cycles into the producing reservoir - a process known as Water-Alternating-Gas (WAG) injection, is miscible in oil, altering

the interfacial tension and causing the oil droplets to swell and become more mobile in the reservoir (Gu & Yang, 2004). This, in combination with other inherent rock properties such as wettability and capillary pressure, control the amount of additional recovery obtained by $CO_2$ flooding, which may range from 8–16% of the original oil in place (Christensen et al., 2001; Rogers & Grigg, 2001)

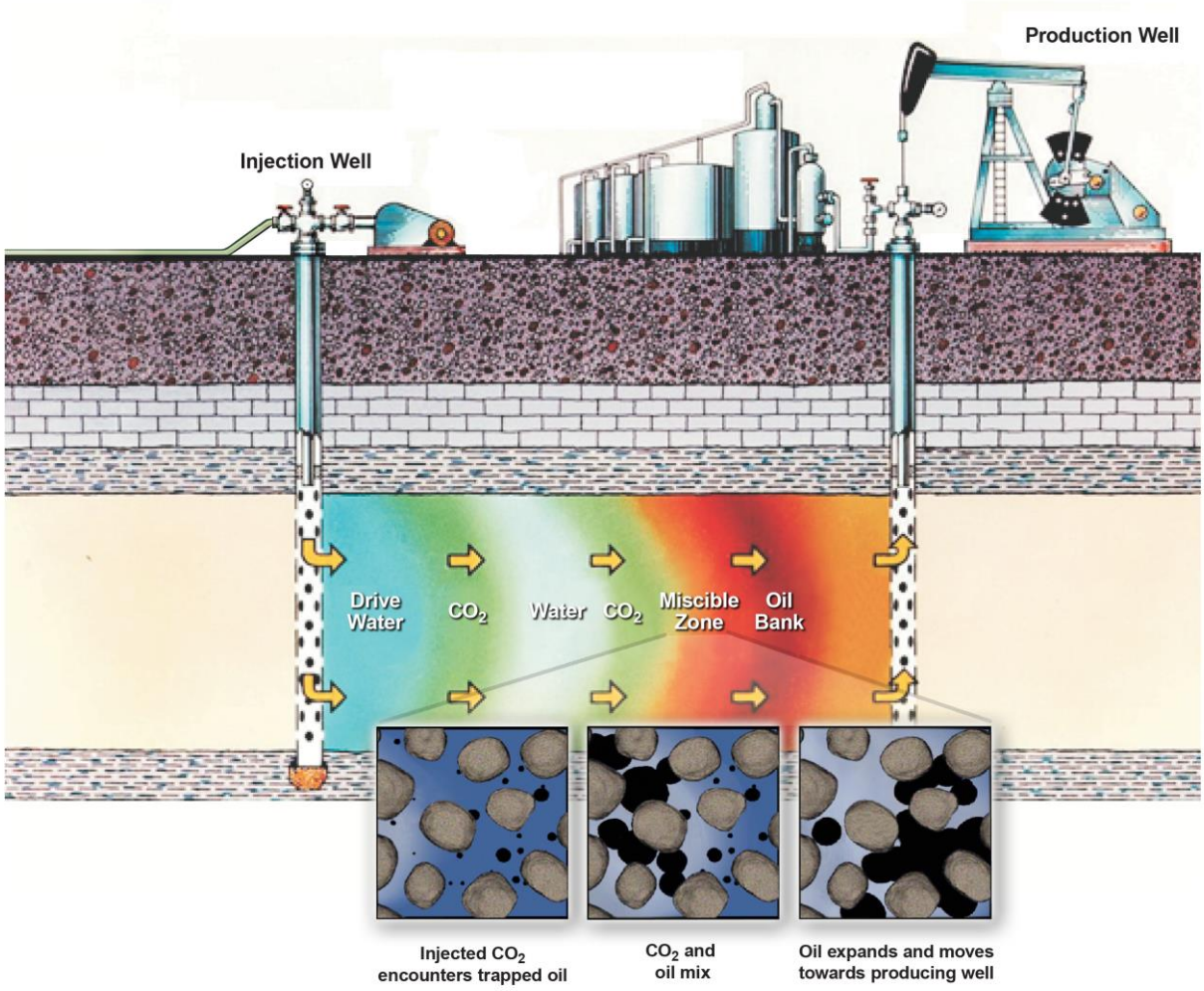



*Figure 3: Illustration of WAG $CO_2$ -EOR, extracted from NETL $CO_2$ primer.*

Injection of $CO_2$ for EOR purposes is not viewed as a fully closed loop sequestration system because some of the $CO_2$ is dissolved in the oil and is produced back at surface as illustrated in figure 3. Laboratory Studies have shown that in WAG, carbonated water injection (CWI) and their respective variants, $CO_2$ storage may range from 5-65% (Ajoma et al., 2021).

### 2.1.2 Sequestration in Saline Aquifers

The consideration for storage of $CO_2$ in deep geological saline aquifers are the results of effort to have a fully closed loop sequestration system. Saline aquifers have very large storage potential and may hold $CO_2$ emissions for decades (McPherson & Cole, 2000). The method of $CO_2$ storage in saline aquifers is by dissolution or mineralization (preferred) due to rock-fluid interactions in carbonate aquifers (Tarrahi & Afra, 2015). Mineralization serves as a very effective way of storing $CO_2$ since it remains in solid state, however, concerns around formation damage in heterogeneous carbonate aquifers due to pore space plugging by precipitated calcium carbonates may limit practical application (Mohamed & Nasr-El-Din, 2012). In deploying large scale sequestration projects in saline aquifers, pressures must be consistently measured especially for bounded reservoirs. This is primarily because as $CO_2$ is injected, the pressure in the reservoir builds, and if it goes above the fracture gradient of the cap rock, then the geologic seals fracture, and $CO_2$ starts to leak into underground sources of drinking water (USDW) or may seep up to the surface (Achanta et al., 2012). Sequestration induced seismicity is also a concern and may be assessed through probabilistic methods before project implementation (Burghardt & Appriou, 2021; Ochie et al., 2022).

## 2.2 Sequestration Infrastructure

### 2.2.1 $CO_2$ Capture Technology

Large sources, such as fossil fuel power plants, fuel processing plants, and other industrial facilities are the primary targets for $CO_2$ capture as capturing $CO_2$ directly from smaller sources in the transportation and residential sectors is anticipated to be more challenging and costly (IPCC, 2005)

IPCC in their 2005 report on Carbon Dioxide Capture and Storage detail 4 basic systems for capturing $CO_2$ from use of fossil fuels:

1. ***Industrial process stream capture*** – This is a long-standing process that has been ongoing for decades and involves $CO_2$ capture as part of chemical processes such as natural gas treatment and ammonia manufacturing (Kohl & Nielsen, 1997). Koch industries in collaboration with Anadarko Petroleum capture $CO_2$ from the Enid fertilizer (ammonia) plant and transport it to the Purdy field in Oklahoma for $CO_2$ EOR purposes (Callahan et al., 2014).
2. ***Post-Combustion stream capture*** – Involves capture of flue gas that are byproducts of fossil fuel combustion and may be achieved using chemical sorbent process. They have current applications in coal and natural gas power plants but may be more efficiently applied to supercritical pulverized coal fired plants and natural gas combined cycle (NGCC) plants (IEA, 2005).
3. ***Oxy-fuel combustion capture*** – Here, high purity $O_2$ is used for combustion rather than air, this results in the production of $CO_2$ and $H_2O$ as the major by-products. Doing this ensure higher purity $CO_2$ being produced from the system making it easier to separate and

capture. This technology may also be applied to plants utilizing fossil fuel as power source if cost effective oxygen separation from air is available.

4. ***Pre-combustion capture*** – Technology is more complex than Post and Oxy-fuel combustion in that it involves generation of a synthetic fuel gas (which contains CO and $H_2$) by reacting the fossil fuel and oxygen. The CO then reacts with steam to form $CO_2$ and hydrogen and the $CO_2$ is separated downstream using physical or chemical processes. A major by-product of these systems are hydrogen rich fuels which can be used in running large industrial plant equipment. There is current application of Pre-combustion capture in Integrated Gasification Combined Cycle power (IGCC) plants, however, IGCC power plants are not common.

These capture technologies (including future technologies being developed) and their applications to different industry types lead to variability in estimating $CO_2$ capture costs for project economic analysis. Researchers at the Great Plains Institute (GPI) published a white paper (Abramson et al., 2020) summarizing near and medium term facilities in the US and their estimated cost ranges.

Table 1: Summary of near- and medium-term facilities, capture targets and cost estimates, source: (Abramson et al., 2020)

| Industry | Number of Facilities | Estimated Capturable $CO_2$ mmt/year | Share of Total Capturable Estimate | Average Estimated Cost $/ton | Range of Cost Estimates $/ton |
|---|---|---|---|---|---|
| Coal Power Plant | 58 | 143.4 | 40.1% | $56 | $46 - $60 |
| Gas Power Plant | 60 | 67.9 | 19.0% | $57 | $53 - $63 |
| Ethanol | 150 | 50.6 | 14.1% | $17 | $12 - $30 |
| Cement | 45 | 32.7 | 9.1% | $56 | $40 - $75 |
| Refineries | 38 | 26.5 | 7.4% | $56 | $43 - $68 |
| Steel | 6 | 14.6 | 4.1% | $59 | $55 - $64 |
| Hydrogen | 34 | 14.4 | 4.0% | $44 | $36 - $57 |
| Gas Processing | 20 | 4.5 | 1.3% | $14 | $11 - $16 |
| Petrochemicals | 2 | 1.7 | 0.5% | $59 | $57 - $60 |
| Ammonia | 3 | 0.9 | 0.3% | $17 | $15 - $21 |
| Chemicals | 2 | 0.7 | 0.2% | $30 | $19 - $40 |
| **Grand Total** | **418** | **357.8** | **100.0%** | **$39** | **$11 - 75** |

### 2.2.2 Transport Alternatives

There are several alternatives for transporting $CO_2$ from source to sink including pipeline transportation (typically in gas or supercritical phase) and transport with trucks, railway, or ships after $CO_2$ has gone through liquefication process. For most large-scale $CO_2$ sequestration projects, pipelines will be the most cost-effective means of transport and for the rest of this study, $CO_2$ transport will simply refer to pipeline transportation.

## 2.3 Sequestration Economics

Whilst many manufacturing companies would like to curb carbon emissions by improving chemical filtration processes within plant operations, the required technology to achieve this comes at a substantial cost (power related costs + equipment and workforce costs + plant modification downtime related costs). To completely ensure that the chemically removed $CO_2$ is not re-released into the environment at some point in the lifecycle, they must develop utilization means, which permanently transforms $CO_2$ to a state where it cannot vaporize into the atmosphere. (Hepburn et al., 2019) documents 10 utilization pathways for $CO_2$ but suggests that only $CO_2$-EOR, concrete manufacturing, bioenergy CCS, and enhanced weathering provided fully or partial closed pathways for storage. When $CO_2$ is not being utilized for any of these processes that typically generate some form of revenue, the only reliable form of safely and permanently removing $CO_2$ from the atmosphere is through CCS via sequestration in saline aquifers.

A significant constraint in creating extensive pure $CO_2$ sequestration projects is the financial aspect of capturing $CO_2$ at the emission source, constructing efficient and affordable transport pipelines (for land-based transportation), injecting it into new or existing wells in saline aquifers, and monitoring for potential leakages due to geological events during injection periods. These capital-

intensive processes must be executed while understanding that the project will not generate direct revenue, and operators must depend on societal benefits and government incentives to ensure success.

### 2.3.1 Role of Government Incentives

The United States introduced carbon tax credits in section 45Q of the United States Internal Revenue Code (26 U.S.C. § 45Q) in 2008 as a means to incentivize rapid adoption and implementation of CCS projects within the country. Since its introduction, only 16 CCUS projects have been implemented and this was primarily due to the relatively small tax credits offered, which in most cases could not offset running costs. Majority of these projects are related to commercialized removal of $CO_2$ at gas processing facilities or fertilizer manufacturing plants and utilization in $CO_2$ EOR projects (Callahan et al., 2014).

In 2018, the government acknowledged the inadequacy of the tax credits provided and revised them, granting up to $35 in tax credits per metric ton of $CO_2$ used in manufacturing with closed pathway storage or geologically stored through EOR projects. Additionally, they offered up to $50 per metric ton of $CO_2$ for geological storage not used for EOR, signifying pure sequestration. Accessing those credits came with strict requirements – to qualify, an operator must meet one of the following criteria:

*Emission source eligibility*

- Capture at least 500,000 $mTCO_2$/yr for power plants.
- Capture at least 100,000 $mTCO_2$/yr for other industries.

*Storage and Utilization Requirements*

- Captured $CO_2$ must be injected into underground geologic formations and permanently sequestered. This includes injection into oil and gas reservoirs (EOR projects), deep saline formations and coal bed seams.
- At least 25,000 $mTCO_2/yr$ must be utilized and permanently fixed in a commercial product. Eligibility is dependent on a life cycle analysis to ensure carbon is permanently stored and not emitted by the same commercial product.

With the 2018 increase, small scale sequestration projects with source and sinks in relatively close proximity became cost effective, however much larger scale projects involving building pipelines across county lines in mid-sized US states were still uneconomical (DaneshFar et al., 2021).

In 2022, the United States government again amended the 45Q tax credits, offering a 43% increase in tax credits for carbon stored via $CO_2$-EOR projects (from $\$35/tCO_2$ to $\$50/tCO_2$), a 70% increase tax credits for pure sequestration ($\$50/tCO_2$ to $\$85/tCO_2$) and up to $\$180/tCO_2$ for direct air capture with storage via pure sequestration. Additionally, eligibility capture thresholds were lowered significantly, point source capture requirements are now a minimum of 12,500 $mTCO_2/yr$ with 18,750 $mTCO_2/yr$ for power plants and 1,000 $mTCO_2/yr$ for direct air capture.

These recent changes mean that most projects that were deemed commercially unviable with unit total sequestration project costs of less than $\$15/tCO_2$ could now break-even and perhaps make significant profits.

## 2.4 Network Optimization of $CO_2$ Sequestration

Medium to large scale carbon sequestration projects involve multiple point $CO_2$ source locations and sinks (which may be geological fields or wells within the fields), and the key questions that must be answered when planning these projects are (1) How to represent pipeline construction costs on a geographical surface (2) what are the possible pipeline routes for transportation that connect sources to sinks? (3) what are the optimal transport routes that minimize overall sequestration costs?

### 2.4.1 Representing Pipeline Routing and Construction Costs on A Geographic Surface

The first question poses a geographical network challenge as it requires Geographical information Systems (GIS) representation. Ideally, the shortest path connecting any 2 points by distance is a straight line and whilst the distance between source and sink is a major factor in this allocation, other social, environmental, and engineering costs must be considered. Some of the key factors that must considered when planning pipeline routes include:

- Terrain Topography
- Population Density
- Neighboring settlements
- Presence of barriers, e.g., streams and rivers
- Existing Right of Way

The Least Cost Path (LCP) can be defined as the most cost-effective path from a start point to a destination and LCP analysis allows for definition of costs associated with movement along a path

based on GIS. Early application of LCP analysis saw researchers combining viewshed information from digital elevation models to determine scenic, strategic, hidden and withdrawn paths for military and environmental planning (Stucky, 1998), with extensions to roadway planning (Yu et al., 2003). Other researchers utilized data gathered from radio tracking of hedgehogs to determine LCP and investigated variations of cost surface values with least cost habitat graphs (Driezen et al., 2007; Rayfield et al., 2010).

Development of surface cost graphs and the determination of LCP are based on concepts relating to graph theory. In practice, when rasterized surface cost graph has been made available, Dijkstra's shortest path algorithms (Dijkstra, 1959) are run to find the LCP between any 2 geo-locations, this fundamentally always return the LCP (which may be non-unique) provided the graph edge weights are non-negative.

Raster-based cost surfaces calculate edge weights by combining social and environmental factors, accounting for minimized distance between node pairs (Hopkins, 1973). The queen's kernel, commonly used in generating these cost surfaces balances computational speed and proximity distortion (Huber & Church, 1985), however, accurate weighing of cell barriers is crucial, especially when designing for pipeline infrastructure (Lugschitz, 2017).

***CostMAP***

In 2019, researchers from Los Alamos National Lab and Montana State University introduced an open-source software package called the Cost Surface Multi-Layer Aggregation Program (CostMAP) for developing these rasterized cost surfaces. CostMAP utilized available pre-

processed GIS data on land cover, slope, population density, natural and man-made barriers and pipeline networks right of way to build weighted cost graphs (Hoover et al., 2019).

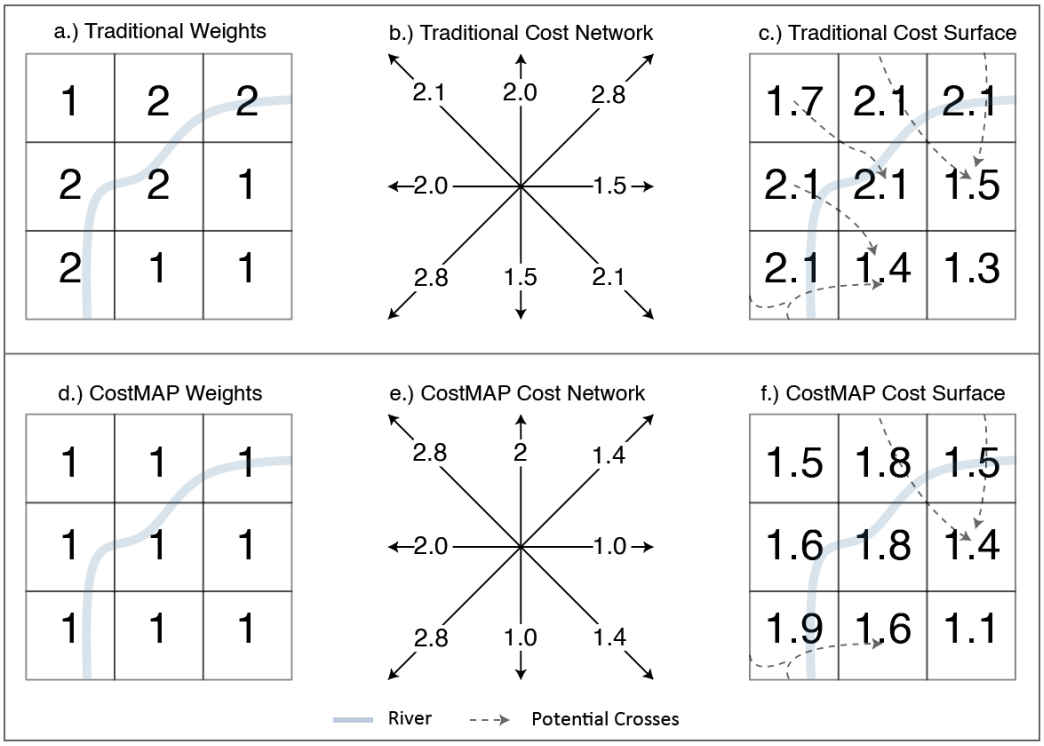



*Figure 4: An example of aggregated cell weight generated using the traditional Queen's kernel (a-c) vs using CostMAP (d-f) source: (Hoover et al., 2019)*

CostMAP offers advancements over traditional methods in the computation of edge weights when barriers such as rivers or roads are present by more accurately identifying these barriers and corridors. Instead of simply treating a cell as a crossing or a barrier if any part of the cell contains a linear feature, CostMAP uses a fine-scale raster to carefully check for actual crossings or corridors within the cell. This approach prevents the overestimation of costs incurred by barriers and helps find feasible routes that other methods might not identify. By using Major and Minor Cells, CostMAP refines the detection of actual crossings between adjacent cells, leading to a more precise estimation of the edge weights in the presence of barriers, this is highlighted in figure 4.

The output of CostMAP is a weighted-cost network and an aggregated cost raster, which can be used for LCP calculations in any GIS-aware software.

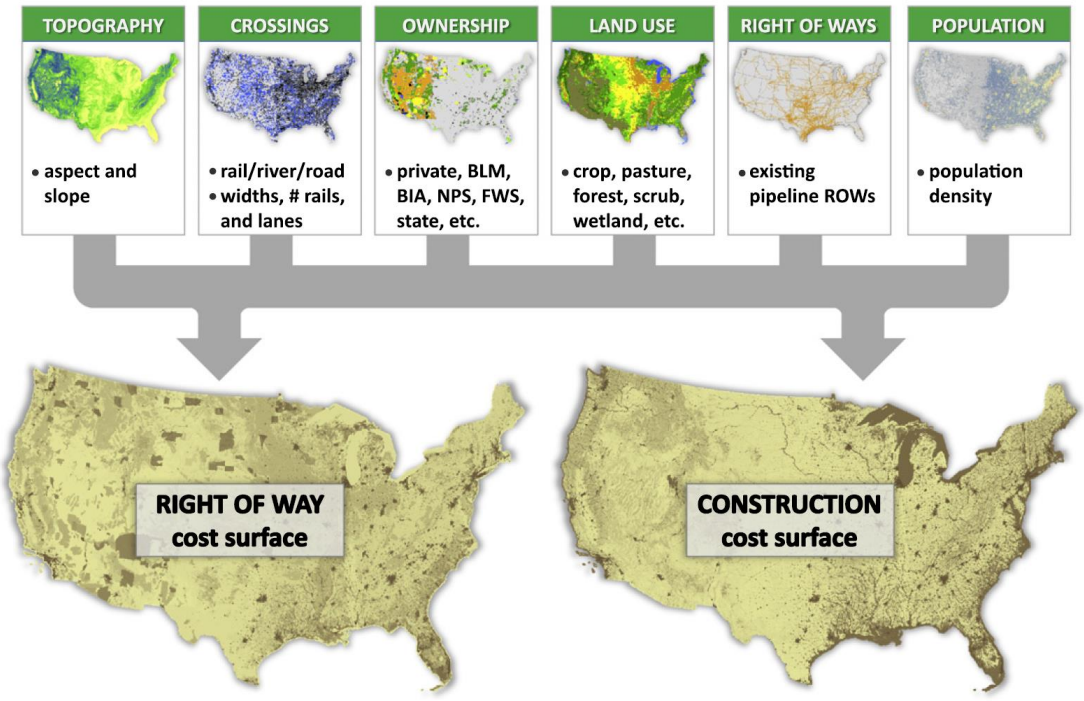



*Figure 5: Generating Right of Way (ROW) and Construction cost surface graphs (Middleton et al., 2012)*

In 2022, a newer version called CostMAP$^{PRO}$ was introduced to majorly address scale challenges for large scale sequestration projects (Talsma et al., 2022). The software produces higher resolution routing surfaces, ranging from 90m to 720m, with the ability to model routes at 10m. Additionally, CostMAP$^{PRO}$ can separate routing and construction cost weights, offering greater flexibility in avoiding specific features without affecting construction costs. Finally, CostMAP$^{PRO}$ allows users to input custom weights to create tailored cost surfaces, enhancing its adaptability to various stakeholder needs, such as avoiding river crossings.

### 2.4.2 Generating Alternate Pipeline Transport Networks

Alternate pipeline transport network generation involves determining a network of LCP that connects a set of sources to sinks, such that a path exists connecting every source to every sink. (Middleton, Kuby, et al., 2012) outlined 5 steps for generating alternate pipeline transport networks (called candidate networks): (1) Define or Import a cost surface, from CostMAP or similar tools (2) Extract Raster LCP through shortest path algorithms like Dijkstra's, (3) Raster to vector conversion which essentially reduces LCPs to a set of discrete nodes and arcs, (4) Removing redundancy caused by duplicate edges along identical cost paths as illustrated in figure 6, and (5) Network refining which involves algorithms defined to further simply the network, by say, collapsing triangles and merging nodes, figure 7.

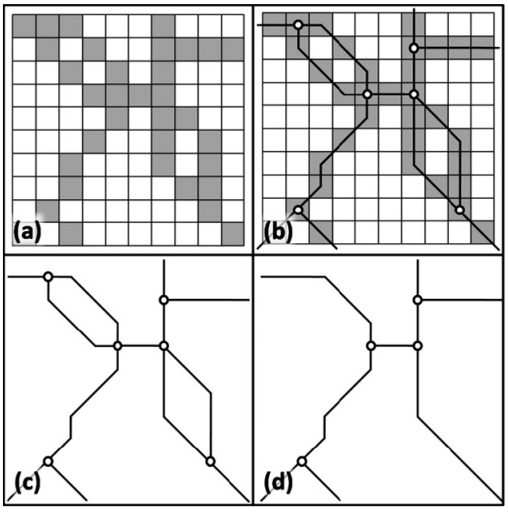



*Figure 6: The candidate network methodology's process for converting raster-to-vector is depicted in these steps. In part (a), the cost surface forms raster paths or corridors, while parts (b) and (c) demonstrate the extraction of a vector network from these paths. Finally, part (d) shows how to strip two or more paths of the same cost between two nodes by removing arcs and nodes. Source (Middleton, Kuby, et al., 2012)*

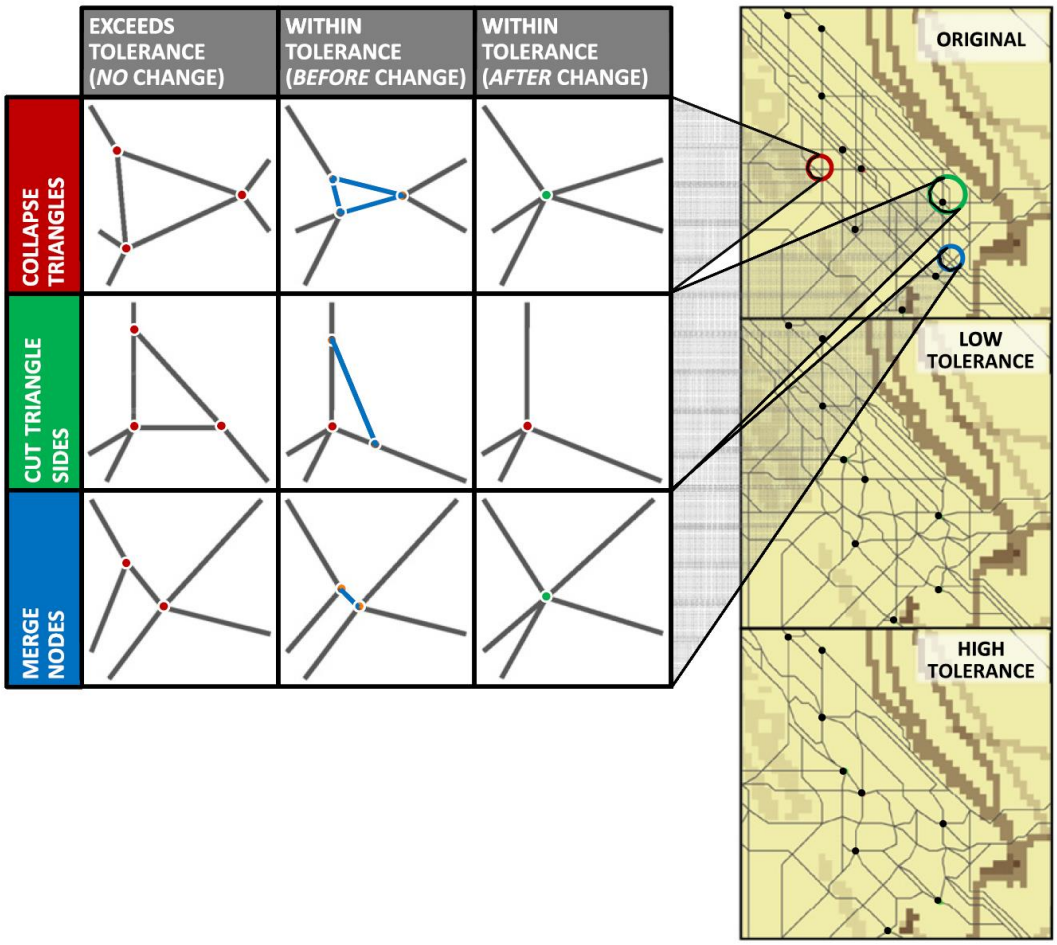
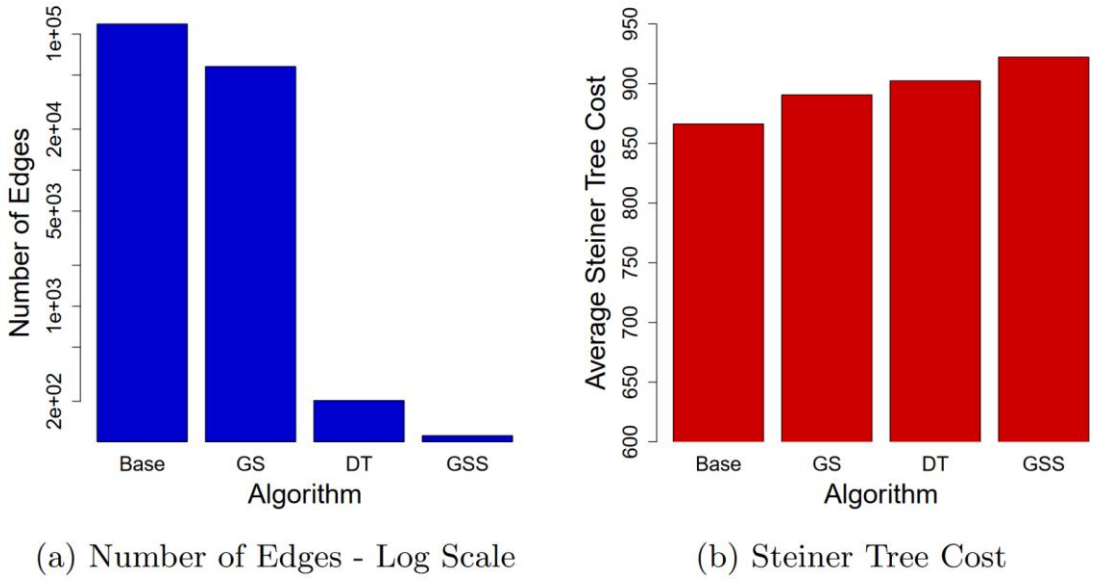



*Figure 7: Shown on the left are three methods for refining the network, along with their effects on the candidate network depicted on the right. As the tolerance level is raised, the candidate network undergoes further refinement, resulting in a decrease in the number of nodes and arcs (Middleton, Kuby, et al., 2012)*

In connecting sources and sinks on a large graph with several edges, many routes may exist and if one desires to get routes between several pairs of nodes, the problem may become intractable or computationally inefficient. (Yaw et al., 2019) while researching on efficient network design proposed the Greedy Subset Spanner (GSS) algorithm as a means of designing provably optimal pipeline transport routes. In their paper, they noted that GSS could reduce the number of edges from a base graph by over 99.9% (figure 8) whilst generating alternate routes with a cost increase

of only ~6% over base Steiner trees which are the most optimal. The use of Delaunay Triangulation (DT), (Delaunay, 1934) to generate set of node pairs for which cost paths can be determined has proven an effective simplified approach to solving this problem, however because of the geospatial nature of the pipeline network generation problem, the distance guarantee of at most 2.418 times the Euclidean distance on any path between 2 points does not hold (Keil & Gutwin, 1992).




(a) Number of Edges - Log Scale (b) Steiner Tree Cost

*Figure 8: Performance metrics for Base Steiner Trees, Greedy Spanner (GS), Delaunay Triangulation (DT) and Greedy Subset Spanner (GSS) (Yaw et al., 2019)*

Although the GSS algorithm offered significant improvements, DT came in a close second with similar edge reduction and better costs. Following the 5-step workflow and using DT in step 2 can lead to the generation of provably optimal candidate networks as illustrated in figure 9 a-c.

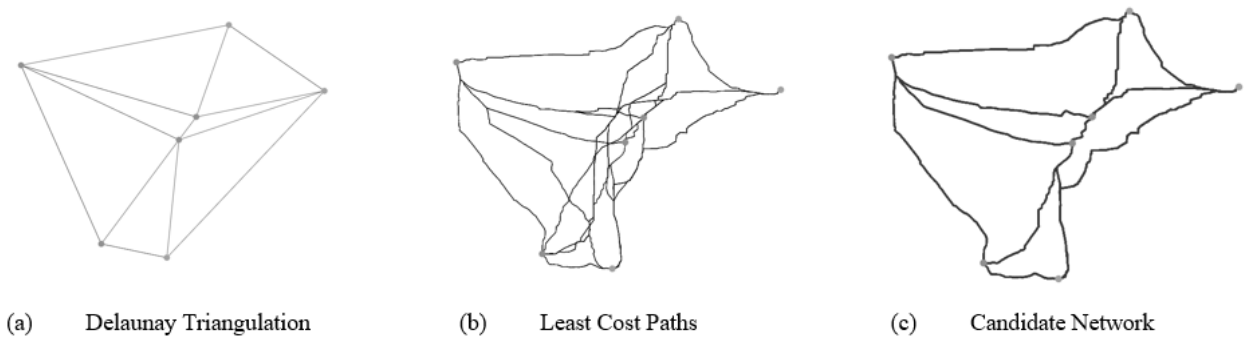



*Figure 9: This illustration showcases the LCP analysis performed by SIMCCS$^{2.0}$. To generate an idealized Candidate Network for CCS pipelines, SIMCCS$^{2.0}$ employs a combination of Delaunay triangles, as shown in (a), to determine the best spatial configuration. Additionally, SIMCCS$^{2.0}$ uses a cost surface and Dijkstra's algorithm to calculate the least cost path across the surface, as seen in (b), in order to create the Candidate Network, as shown in (c).*

### 2.4.3 Determination of Optimal Transport Routes

The next step after generating the alternate pipeline routes based on the cost surface graph, Delaunay triangulation, and Dijkstra's shortest path algorithm, is the formulation and implementation of a mathematical optimization model to find optimal transport routes and minimize overall $CO_2$ sequestration costs. The Overall optimization problem is stated below:

Given:

- A set of $CO_2$ capture sources with associated capture costs in $ per metric ton of $CO_2$ captured and total $CO_2$ capture capacity.
- A set of $CO_2$ storage locations with associated storage or injection costs in $ per metric ton of $CO_2$ injected and total $CO_2$ storage capacity.
- A set of alternate pipeline routes with associated construction costs in $ per metric ton of $CO_2$ transported.

- Possible existing pipeline routes with zero associated construction cost, fixed volume capacity limits and transportation costs in $ per metric ton of $CO_2$ transported.
- A target $CO_2$ capture amount.

Find the minimum cost network that ensures $CO_2$ target capture amount is met, honoring the following constraints:

1. $CO_2$ captured and stored must be less than or equal to the capture and storage capacities.
2. Existing pipeline capacity limits are not violated.
3. Flow is unidirectional along existing pipeline.

### 2.4.4 Mathematical Model Formulation

Translating the description to mathematical terms which can be used as a basis for building the $CO_2$ optimization model is trivial since an existing formulation was developed and implemented in SimCCS (Middleton et. al, 2020). The mathematical model, which borrows its notations from SimCCS is given below:

**Sets**

| | |
|---|---|
| *S* | *Set of all $CO_2$ capture source nodes.* |
| *R* | *Set of all $CO_2$ sinks or injection sites nodes.* |
| *N* | *Set of all Transshipment nodes.* |
| *I* | *Set of all nodes.* |
| *C* | *Set of pipeline trends.* |
| *A* | *Set of all nodes to node arcs representing alternate and existing pipelines.* |
| P $\epsilon$ A | *Set of all existing pipeline arcs.* |

**Parameters**

| | |
|---|---|
| $Q_i^S$ | *$CO_2$ annual capture capacity at source i ($MtCO_2$/yr).* |
| $Q_j^R$ | *$CO_2$ storage capacity at sink j ($MtCO_2$).* |
| $F_i^S$ | *Fixed capture cost of $CO_2$ at source i ($M).* |
| $F_j^R$ | *Fixed storage cost of $CO_2$ at sink j ($M).* |
| $V_i^S$ | *Variable capture cost of $CO_2$ at source i ($/t$CO_2$).* |
| $V_j^R$ | *Variable capture cost of $CO_2$ at sink j ($/t$CO_2$).* |
| $Q_{ac}^{max}$ | *Maximum annual capacity of pipeline arc a with trend c ($MtCO_2$/yr).* |
| $Q_{ac}^{min}$ | *Minimum annual capacity of pipeline arc a with trend c($MtCO_2$/yr).* |
| $\alpha_{ac}$ | *Transportation cost of pipeline arc a with trend c ($/t$CO_2$).* |
| $\beta_{ac}$ | *Build cost of pipeline arc a with trend c ($M/yr).* |

**Decision Variables**

| | |
|---|---|
| $s_i \in \{0,1\}$ | *Binary variable indicating if a source i is activated.* |
| $r_j \in \{0,1\}$ | *Binary variable indicating if a sink j is activated.* |
| $y_{ac} \in \{0,1\}$ | *Binary variable indicating if a pipeline arc a with trend c is built.* |
| $a_i \in \mathbb{R}$ | *The amount of $CO_2$ captured at source i ($tCO_2$/yr).* |
| $b_j \in R$ | *The amount of $CO_2$ stored at sink j ($tCO_2$/yr).* |
| $f_{ac} \in R$ | *The amount of $CO_2$ flow in pipeline arc a built with trend c ($tCO_2$/yr).* |
| $CO_2T \in R$ | *The target amount of $CO_2$ to be sequestered during project life.* |

The model formulation is a MIP problem and the objective function as described in the previous section is to minimize sequestration cost. Sequestration costs consist of capture cost, storage cost, pipeline build cost and transportation cost. Written mathematically as:

$$\textbf{min} \sum_{i \in S}(F_i^S s_i + V_i^S a_i) + \sum_{j \in R}(F_j^R r_j + V_j^R b_j) + \sum_{a \in A}\sum_{c \in C}\beta_{ac} y_{ac} + \sum_{a \in A}\sum_{c \in C}\alpha_{ac} f_{ac}$$

*capture* *storage* *pipe build* *transportation*

Subject to the following constraints

Arc capacity bounds: $Q_{ac}^{min} \leq f_{ac} \leq Q_{ac}^{max} \quad \forall a \in A, \forall c \in C$

Single direction arc flow: $\sum_{c \in C} y_{ac} \leq 1 \quad \forall\, a \in A$

Flow balance: $\sum_{\boldsymbol{a \in A}} \sum_{\boldsymbol{c \in C}} f_{ac} - \sum_{\boldsymbol{a \in A}} \sum_{\boldsymbol{c \in C}} f_{ac} = 0 \quad if\ n \in N$

Demand balance: $\sum_{\substack{\boldsymbol{a \in A} \\ src(k)=n}} \sum_{\boldsymbol{c \in C}} f_{ac} - \sum_{\substack{\boldsymbol{a \in A} \\ dst(k)=n}} \sum_{\boldsymbol{c \in C}} f_{ac} = -b_n \quad if\ n \in R$

Supply balance: $\sum_{\substack{\boldsymbol{a \in A} \\ src(k)=n}} \sum_{\boldsymbol{c \in C}} f_{ac} - \sum_{\substack{\boldsymbol{a \in A} \\ dst(k)=n}} \sum_{\boldsymbol{c \in C}} f_{ac} = a_n \qquad if\ n \in S$

Capture capacity bounds: $a_i \leq \boldsymbol{Q_i^S} s_i \quad \forall i \in S$

Storage capacity bounds: $b_j \leq \boldsymbol{Q_j^R} r_j \quad \forall j \in R$

Target capture: $\sum_{\boldsymbol{i \in S}} a_i \geq CO_2T$

Pipeline Build cost: $\boldsymbol{C_1} * f_{ac} + \boldsymbol{C_2} = \alpha_{ac} \quad \forall a \in A$ where $\boldsymbol{C_1}$ & $\boldsymbol{C_2}$ are trendline constants

The constants $\boldsymbol{C_1} \in C$ and $\boldsymbol{C_2} \in C$ are based on trend lines used to represent cost per unit distance profiles for given $CO_2$ volume. Representing $CO_2$ transportation costs with linearized trends was proposed in (Middleton, 2013) where pipeline transportation costs were a function of the volume of $CO_2$ being transported. In other publications, (Jones et al., 2022; Whitman et al., 2022), researchers demonstrated the utilization of 2 linear trends that cover 11 distinct pipeline capacities in SimCCS. The generation of these trends are discussed in the next section.

### 2.4.5 Representing $CO_2$ pipeline Construction Costs with Trendlines

Determining the required capital investment for constructing $CO_2$ pipelines is a complex process that involves considering several factors. Pipeline length, expected flow rates, inlet and outlet pipeline pressures, elevation changes, and topography are just a few of the crucial factors that must

be evaluated when assessing pipeline construction costs and, optimizing these costs in sequestration project planning is a non-trivial problem. This challenge led the US Department of Energy (DOE) Office of Fossil Energy and Carbon Management (FECM) to collaborate with the National Energy Technology Laboratory (NETL) to develop the FECM/NETL $CO_2$ Transport Cost Model ($CO_2$_T_COM). This excel-based software, powered by Visual Basic for Applications (VBA) macros, takes input from users, and generates realistic cost breakdowns, including capital and operating costs, based on specific financial and engineering inputs (Morgan et al., 2022)

While $CO_2$_T_COM is useful for in-depth analysis, its output needs to be translated into a format that can be used in the network optimization model. Inputs from table 2 were used to generate total transport costs for different $CO_2$ annual flow rates by (Jones et al., 2022). They derived two linear trends to represent the cost of $CO_2$ in $M per kilometer (figure 10). This cost trends were applied in the SimCCS optimization model, and the costs generated were compared to $CO_2$_T_COM resulting in an acceptable average absolute error (AAE) of 3%.

Choosing a linear approximation instead of a quadratic curve was due to anticipated computational complexity. Mixed-integer linear programming (MILP) which is a branch of mathematics and operations research that focuses on linear optimization, requires constraints and objective functions to be linear; otherwise, the problem becomes a mixed-integer quadratic programming (MIQP) problem, which is harder to solve even with state-of-the-art solvers. Although a lower AAE could be achieved by a quadratic function, the increased solve time is not a desirable tradeoff.

*Table 2: Assumptions for $CO_2$_T_COM transport cost trends used in SimCCS.*

| Input | Values |
|---|---|
| Segment Length | 80km (50 miles) |
| Segment Inlet Pressure | 15 Mpa (2175 psi) |
| Segment Outlet Pressure | 8.6 Mpa (1250 psi) |
| Pumps per 100 miles | 2 |
| Construction Cost Model | PARKER |
| Region | MW (Midwest) |
| Pipeline Capacity Factor | 0.8 |
| Capital Charge Factor | 0.11 |

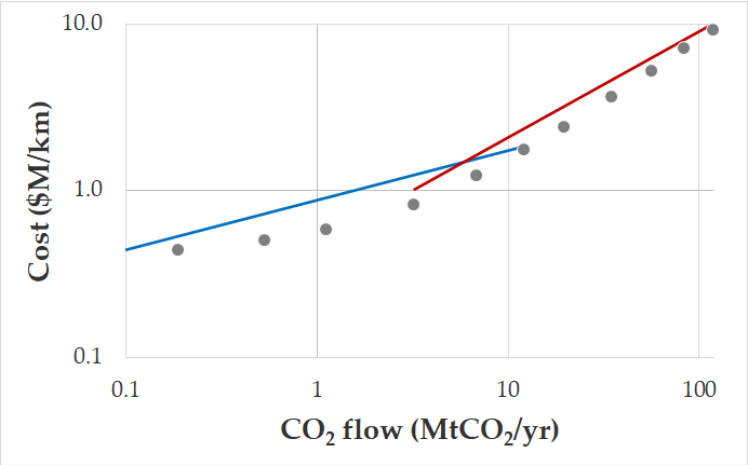



*Figure 10: Linear trend approximation of pipeline cost (Whitman et al., 2022)*

## 2.5 Existing Solutions and Limitations

There are several solutions for optimizing deployment scenarios for CCS projects including JRC InfraCCS (Morbee et al., 2011), GETCO (Gale et al., 2001), however, there has been a wide

adoption of SimCCS (and SIMCCS$^{2.0}$) since release and it has become the industry standard for evaluating CCS projects.

Published use cases of SimCCS in large scale sequestration planning include evaluations carried out in North America, Western Europe and Asia (Bielicki et al., 2014; Middleton & Brandt, 2013; Stauffer et al., 2014). Other evaluations cover utilization with different capture sources types – such as power plants, chemical and oil refining plants (Middleton et al., 2014; Middleton, Keating, et al., 2012), and different sink types apart from saline aquifers and – such as producing sands for $CO_2$-EOR (Middleton et al., 2011), depleted hydrocarbon bearing shales (Bielicki et al., 2018) and, stacked reservoir systems (Ellett et al., 2017). There have also been recent publications on the application of SimCCS to the evaluation of sequestration economics in localized regions with the US, leveraging on carbon tax credits to determine optimal source sink pairings (DaneshFar et al., 2021).

***Limitations***

Limitations reported in literature typically refer to solving large scale sequestration projects that involve multiple sources and sinks spread over a large geographic region. To address these challenges, upscaling cost surface graphs was proposed to increase speed of generating alternate paths and the use of greedy subset spanner (GSS) to find effective albeit less optimal routes have been proposed (Talsma et al., 2022; Yaw et al., 2019). (Lobo, 2017) proposed speeding up the MIP network optimization by adding valid inequalities to strengthen the original SimCCS mathematical formulation.

One Major limitation that has remained unaddressed is embedding existing pipelines in techno-economic optimization tools like SimCCS, and that limitation is addressed by this research.

# CHAPTER 3: Methodology

## 3.1 Translating Geographical Coordinates to Graph Coordinates

The cost surface graphs were obtained from open source published results of CostMAP simulations generated for SimCCS usage and were represented as numbered grid cells with the southwestern corner point and the grid cell spacing given.

To translate these corner points to latitude and longitude coordinates, algorithms were proposed by the developers of SimCCS in their GitHub repository (SimCCS, 2021). The code for this was adapted and utilized in this research and is detailed in "*geotransformations.py*" in the Appendix.

## 3.2 Generating Alternate Transport Routes

### 3.2.1 Delaunay Triangulation

Implementation of Delaunay Triangulation (DT) in Python was done using the Delaunay class in SciPy Python package. First steps involved translating the latitude and longitude of our sources and sinks into positional X and Y locations on our graph surface and this was done using methods outlined in "*geotransformation.py*" in the Appendix. Next those X and Y points are used to generate Delaunay triangles with successive pairs of nodes connected by lines. The pair of nodes are returned as outputs for use in computation of shortest distances. An example of the results of Delaunay triangulation given the following (x, y) points: {(10, 18), (20, 75), (50, 50), (80, 35), (80, 90)} is illustrated in figure 11 below:

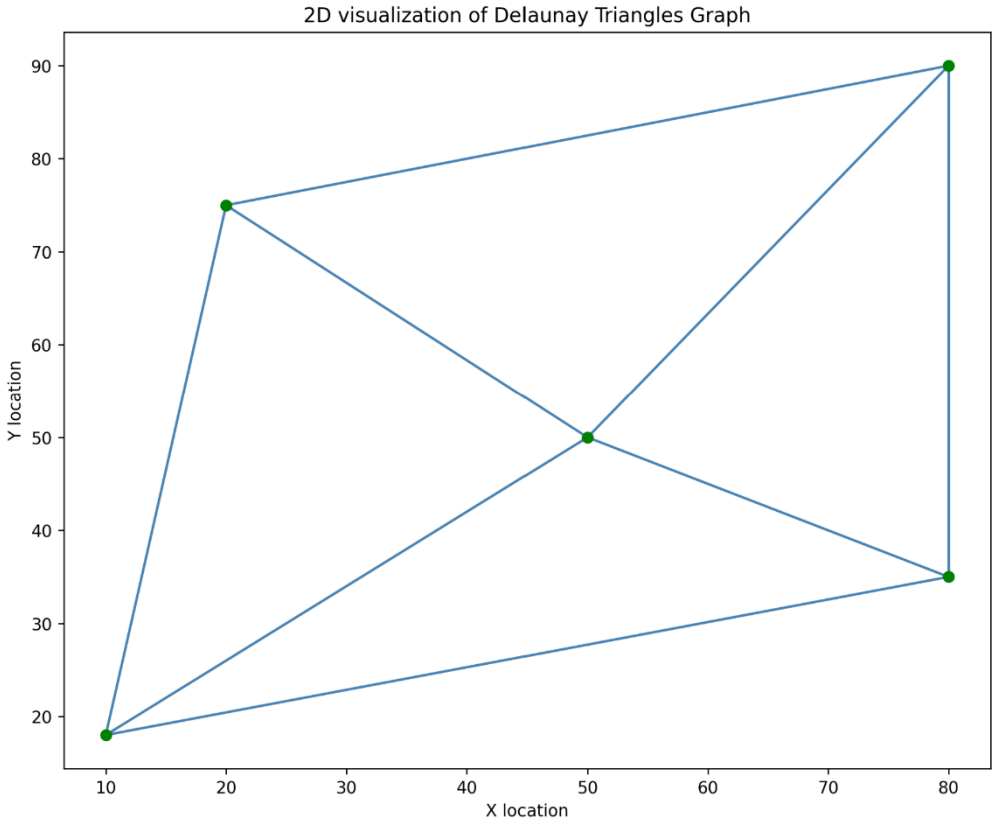



*Figure 11: Plot of Delaunay Triangle generated from specified input data.*

### 3.2.2 Embedding Existing Pipeline Routes

The previous section discusses generating node pairs which will be used to define shortest paths on a cost graph surface assuming that we intend to build new pipelines to kickstart the $CO_2$ sequestration project. While this is useful, one key aspect of alternate routing is how to capture existing pipeline routes within our graph. This is the focus of this research and the proposed implementation details are discussed briefly.

***Proposed Solution and Implementation***

Given that costs associated with transportation of $CO_2$ are determined by the chosen transportation route which is selected based on LCPs on the surface cost graph, the obvious, albeit difficult to implement solution, is to modify the cost graph with zero cost edges along an existing pipeline route.

To achieve this, the following assumptions are made:

1. Long transport pipelines (kilometers or miles in length) are made up of discrete shorter fixed length pipelines with pipe fittings such as elbows, tee-connections, reducers, etc.,
2. The shorter fixed length pipelines are joined together by welding or coupling using fittings.
3. The length of the pipe fittings can be neglected over long pipeline network distances and only the length of short, fixed pipe is considered.
4. The ends of each fixed pipeline represent a physical location with an actual latitude and longitude location; hence the entire length of the pipeline can be represented by a discrete set of Lat-Long points.

The above assumptions allow for, if available, one to directly modify the edges representing the existing pipeline and set their weights to zero. For this to occur however, the exact edges must exist on the surface cost graph, and this may not be practicable for a few reasons:

- **Reason 1**: The shorter pipelines are of fixed length meaning that regardless of orientation, the distance between two ends is the same. Based on the spherical nature of the earth, distances between consecutive grid points may not be the same, hence there will be a mismatch of pipe Lat-Long and grid Lat-Long.

- **Reason 2**: To work around reason one, one may consider generating a very fine-precision cost graph by modifying reducing the grid spacing to say one meter. This work-around becomes impractical due to limitations with graph generation and shortest distance calculations. Generating the cost graph grid points is at least $O(n^2)$ time complexity and the Dijkstra's shortest path algorithm is at least of $O((V + E)log\ V)$ time complexity where V is number of vertices and E is number of edges. This means as the grid spacing reduces, the number of vertices and edges increases, and the computational time increases quadratically on average. Storage space also becomes a huge concern for finer grids, storing a 200 x 400 grid is much more efficient than an 8000 x 9000 grid.

With the limitations described above, coupled with the fact that most operators/researchers do not have an exact set of discrete Lat-Long points along all pipe segments in a pipeline network, a more practicable solution was developed.

Given any existing pipeline network, obtain sparse Lat-Long coordinates representing sections along the pipeline from either the operator or manually using the US National Pipeline Mapping system (NPMS). Each section is assumed to be linear and can be connected by a straight line. Using the geo locations of these sections, approximate x and y grid coordinates can be calculated using methods outlined "*geotransformation.py*" in the Appendix. If an edge exists in the map between successive geolocation grid points, we modify the weight (or cost) of the edge to zero, however if an edge does not exist, we may formulate some.

The process of formulation involves calculation of shortest paths between the two grid points in the graph using weighted edges. Once the grid points and edges along these shortest paths have

been identified, we add those points to the set of pipeline Lat-Long points and set the edge weights (or costs) to zero.

An algorithm designed to implement this is shown below:

***ALGORITHM TO ADD EXISTING PIPELINE TO NETWORK GRAPH***

---

***Step 1****: Convert discrete pipeline lat-long points to edges*
*Input: loc_points, a set of discrete pipeline lat-long points*
*Output: loc_pair, a list of edge pairs*

```
Create a new variable: loc_pair = list()
for i in range(len(loc_points) - 1):
    loc_pair.append((loc_points[i], loc_points[i+1]))
end for
Find cell location of loc_pair points and assign to cell_pair using findCell function
for i in range(len(loc_pair)):
    cell1 = findCell(loc_pair[i][0])
    cell2 = findCell(loc_pair[i][1])
    cell_pair.append((cell1, cell2))
end for
```

***Step 2****: Modify cell_pair to include edges from shortest paths algorithm*
*Input: cell_pair, a list of edge pairs; edges, a set of edges in G(V, E)*
*Output: cell_pair_mod, a modified list of edge pairs*

```
Create a new variable: cell_pair_mod = list()
forall nodepair in cell_pair:
    if nodepair in edges:
        cell_pair_mod.append(nodepair)
    else:
        path = shortest_path(source=nodepair[0], destination=nodepair[1])
        forall edge in path:
            cell_pair_mod.append(edge)
        end for
    end if
end for
```

***Step 3****: Modify graph G(V, E) and set all edge weights for edges in cell_pair_mod to zero*
*Input: G(V, E), a cost surface graph; cell_pair_mod, a modified list of edge pairs*
*Output: G'(V, E'), a modified cost surface graph with some edge weights set to zero*

```
forall nodepair in cell_pair_mod:
    G.edges[nodepair][weight] = 0
end for
```

---

### 3.2.3 Tie-in Points – Calculate or Assign

Once the graph cost surface has been modified with zero weight edges to represent existing pipelines, one key concern becomes how to limit entry and exit points from pipelines. The major reason this concern arises is that, since a zero-weight path exists in the graph, finding the shortest path between any two locations within the vicinity of the pipeline may lead to multiple entry and exit points around the pipeline. This is illustrated in figure 12 below. In 12(a), the shortest paths between two location pairs are shown if no pipeline (zero cost path) exists, a zero-cost pipeline is introduced in 12(b) and in 12(c), we see how the shortest path changes once the pipeline has been added. We also see multiple entry and exit points along a single pipeline.

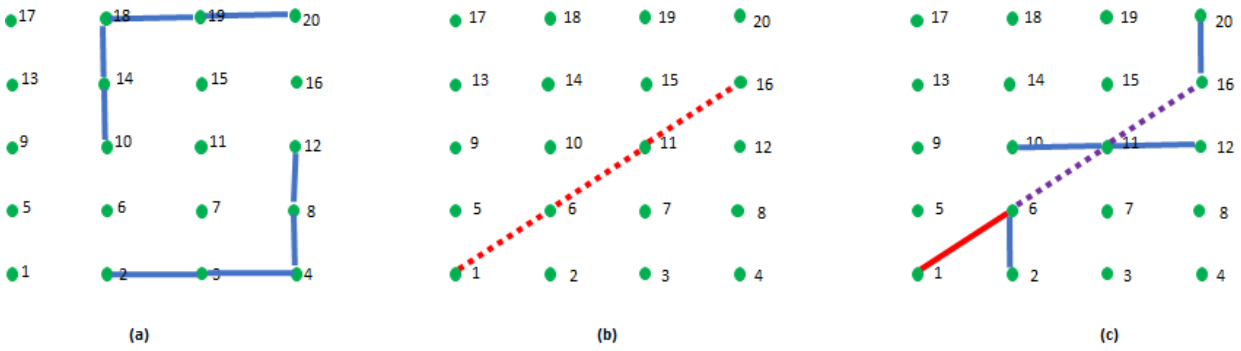



*Figure 12: Illustration of effect of adding zero cost path to LCP generated by Dijkstra's Algorithm. In (a) the zero-cost path does not exist, and the blue paths represent the LCP, (b) shows the introduction of a zero-cost path in red and (c) shows the new LCP generated which utilizes the zero-cost path.*

In practice, having multiple entry and exit points along a single pipeline is usually unwanted as pressure at entry points must be level or graded properly to discourage backflow of fluid from entry point with higher pressure to entry point with lower pressure. Doing this requires installation of compressors and pressure instrumentation to monitor pressure levels along the pipeline consistently, all of which are expensive operations.

Assuming an operator has decide to allow only a single tie-in entry location on an existing pipeline, allowing $CO_2$ from a nearby capture sites to be routed through pipeline, and this same operator is open to having multiple exit locations where $CO_2$ will be routed to storage site, the question then arises - *should the tie-in points be selected based on regional knowledge, land agreements and company preferences, or should it be assigned based on some shortest path calculation?*

There are four scenarios to consider when implementing tie-in points and "exclusion" zones. Exclusion zones represent regions of your pipeline where you want no tie-in location and become practical for pipelines where certain sections are in residential or hilly areas. The possible scenarios are given below and illustrated in figure 13 below:

- Case 1: 2 tie-in points (entry and exit) along a given pipeline, exclusion elsewhere.
- Case 2: 2 tie-in points with exclusion at ends
- Case 3: Singe tie-in point with all exclusion but source or sink.
- Case 4: Single tie-in point with exclusion before or after

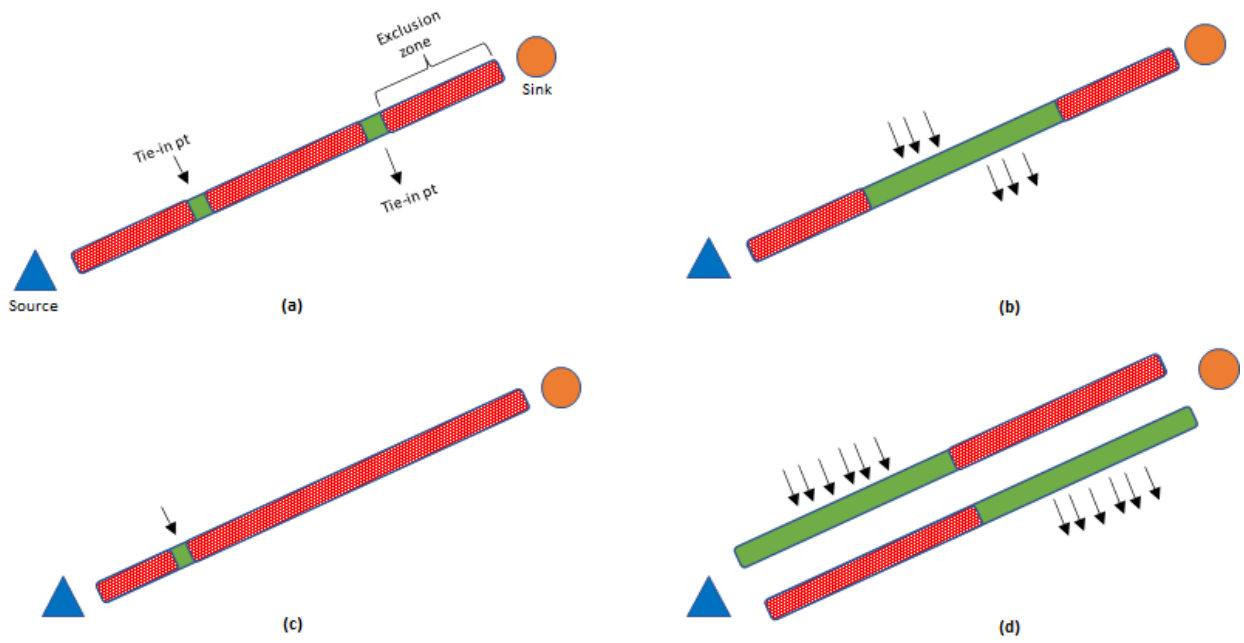



*Figure 13: Illustration of the 4 methods to define tie-in points along an existing pipeline. (a) Case 1: 2 tie-in points are explicitly defined with exclusion everywhere else in pipe, (b) Case 2: 2 tie-in points with exclusion at ends, (c) single tie-in point with exclusion everywhere else but source/sink, (d) Single point with exclusion before or after.*

### *Case 1: 2 Tie-In Points (Entry and Exit) Along A Given Pipeline, Exclusion Elsewhere*

Given two preferred tie in-locations along an existing pipeline, the cost graph can again be modified to ensure that for any optimal $CO_2$ transport network that may be designed utilizing segments of the existing pipeline, the preferred tie-in locations are honored. To achieve this, set the edge weight for all inbound and outbound edges from vertices along the pipeline to a large number, all vertices but the ones representing the preferred tie-in locations. This is done to discourage the shortest path algorithms from using those points during generation of alternate routes. The Algorithm to achieve this is as follows:

***Algorithm 1***

---

***Input**: G(V, E) a cost surface graph; P(V), a list of graph vertices along an existing pipeline; tie_points, a list containing geolocation of 2 preferred tie-in points on existing pipeline*
***Output**: G'(V, E'), a modified cost surface graph with some edge weights along pipeline vertices set to 1e9*

```
Create A list: tie_vertices = list()
forall nodepair in tie_points:
   cell = findCell(nodepair)
   tie_vertices.append(cell)
end for

forall edges in G.edges:
   #in
   if (edges[1] in P) and (edges[0] not in P) and (edges[1] != tie_vertices[0]) and (edges[1] != tie_vertices[1]):
      G.edges[edges][weight] = 1e9
  end if
   #out
   if (edges[0] in P) and (edges[1] not in P) and (edges[0] !+ tie_vertices[0]) and (edges[0] != tie_vertices[1]):
      G.edges[edges][weight] = 1e9
  end if
end for
```

---

***Case 2:  2 Tie-In Points with Exclusion at Ends***

In this case, as illustrated in figure 13(b) above, you want to exclude regions of the pipeline before and after certain tie-in locations. The result of this step gives you an open segment of the pipeline where any two tie-in points may exist, and the exact locations will be determined by the shortest path algorithms. To implement this, a slight modification is made to Algorithm 1, and the nodes along the pipeline before and after tie-in point indexes have their in and outbound edge weights increased as shown in Algorithm 2 below:

***Algorithm 2***

---

***Input**: G(V, E) a cost surface graph; P(V), a list of graph vertices along an existing pipeline; tie_points, a list containing geolocation of 2 preferred tie-in points on existing pipeline*
***Output**: G'(V, E'), a modified cost surface graph with some edge weights along pipeline vertices set to 1e9*

```
Create A list: tie_vertices = list()
forall nodepair in tie_points:
   cell = findCell(nodepair)
   tie_vertices.append(cell)
end for

Create A list: exclusion = list()
idx_1 = P.index(tie_vertices[0]) #index location of point1
idx_2 = P.index(tie_vertices[1]) #index location of point2
exclusion = P[:idx_1] + P[idx_2:] #slice P and get points before and after tie points on both sides

forall edges in G.edges:
   #in
   if (edges[1] in exclusion) and (edges[0] not in P) and (edges[1] != tie_vertices[0]) and (edges[1] != tie_vertices[1]):
      G.edges[edges][weight] = 1e9
  end if
   #out
   if (edges[0] in exclusion) and (edges[1] not in P) and (edges[0] != tie_vertices[0]) and (edges[0] != tie_vertices[1]):
      G.edges[edges][weight] = 1e9
  end if
end for
```

---

***Case 3:*** **Singe Tie-In Point with All Exclusion but Source or Sink**

This case models a scenario where an operator wants to have only one tie-in point on the pipeline, this is because on either end of the pipeline there is an existing sink you want to feed $CO_2$ into. Assuming the pipeline has additional transport capacity, that location on the pipeline can be used

to tie-in extra $CO_2$ sources. This is illustrated in figure 13(c) and the algorithm is based on a modification of algorithm 2 presented above.

***Algorithm 3***

---

***Input**: G(V, E) a cost surface graph; P(V), a list of graph vertices along an existing pipeline; tie_points, a list containing geolocation of 1 preferred tie-in point on existing pipeline*
***Output**: G'(V, E'), a modified cost surface graph with some edge weights along pipeline vertices set to 1e9*

```
Create A list: tie_vertices = list()
forall nodepair in tie_points:
    cell = findCell(nodepair)
    tie_vertices.append(cell)
end for

Create A list: exclusion = list()
exclusion = P[:-1] #assuming source/sink is at end of pipeline OR
exclusion = P[1:] #assuming source/sink is at beginning of pipeline

forall edges in G.edges:
    #in
    if (edges[1] in exclusion) and (edges[0] not in P) and (edges[1] != tie_vertices[0]) and (edges[1] != tie_vertices[1]):
        G.edges[edges][weight] = 1e9
    end if
    #out
    if (edges[0] in exclusion) and (edges[1] not in P) and (edges[0] != tie_vertices[0]) and (edges[0] != tie_vertices[1]):
        G.edges[edges][weight] = 1e9
    end if
end for
```

---

***Case 4:*** **Single Tie-In Point with Exclusion Before or After**

This case solves for the scenario where an operator only wants to exclude a section of a pipeline, to the left or right of a single geolocation; it is a more relaxed version of case 2 as illustrated in figure 13(d). The algorithm to achieve this is as follows:

***Algorithm 4***

---

***Input****: G(V, E) a cost surface graph; P(V), a list of graph vertices along an existing pipeline; tie_points, a list containing geolocation of 1 preferred tie-in point on existing pipeline*
***Output****: G'(V, E'), a modified cost surface graph with some edge weights along pipeline vertices set to 1e9*

```
Create A list: tie_vertices = list()
forall nodepair in tie_points:
    cell = findCell(nodepair)
    tie_vertices.append(cell)
end for

Create A list: exclusion = list()
exclusion = P[:P.index(tie_vertices[0])] #assuming exclusion zone is before tie-in point OR
exclusion = P[P.index(tie_vertices[0])+1:] #assuming exclusion zone is after tie-in point

forall edges in G.edges:
    #in
    if (edges[1] in exclusion) and (edges[0] not in P) and (edges[1] != tie_vertices[0]) and (edges[1] != tie_vertices[1]):
        G.edges[edges][weight] = 1e9
    end if
        #out
    if (edges[0] in exclusion) and (edges[1] not in P) and (edges[0] != tie_vertices[0]) and (edges[0] != tie_vertices[1]):
        G.edges[edges][weight] = 1e9
    end if
end for
```

---

In summary, as a response to the question posed earlier – assigning, or calculating tie-in points, algorithms 1 allows for fully constrained tie-in location assignment while 2 - 4 offers some flexibility with that choice, whilst still allowing an operator to limit selection to preferred regions.

### 3.2.4 Shortest Connecting Path Estimation

Considering that the cost surface graph is simply a directed graph with non-negative edge weights, Dijkstra's algorithm was used in estimating shortest paths. To achieve this, pairs of vertices generated from Delaunay triangulation discussed in section 3.2.1 are used and since distance(v1, v2) = distance(v2, v1), any shortest path generated, and the corresponding weighted cost can be utilized for building bidirectional arcs when running the network optimization.

The open-source Python program, NetworkX, has a custom implementation of Dijkstra's algorithm and since the alternate route candidate network class was built as an abstraction of the Digraph class in NetworkX (shown in *"alternateNetworGeo.py"* in the Appendix), it became redundant to attempt any manual implementation.

Using this library, it is also possible to use bellman-ford algorithm for shortest path estimation, this may become necessary if the operator seeks to further incentivize flow through pipeline and during embedding, sets pipeline graph edge weights to negative values.

### 3.2.5 Solving with Intersecting Shortest Paths in Practice

In practice, generating shortest paths in sequence using vertex pairs generated by the Delaunay triangulation may lead to intersecting paths being generated. This frequently occurs if the $CO_2$

sources or sinks are in proximity and there is a region of the cost surface graph with significantly smaller edge weights. Intersecting paths are undesirable as they require significant engineering work to design - one pipeline must be buried deeper than the other or raised using elbows above the other.

To solve this problem, a diagonal exclusion zone may be generated around alternate pipeline paths after each pass of resulting DT vertices through Dijkstra's algorithm. This diagonal exclusion zone is generated easily by using simple properties of a rectangular grid.

If n is the grid width, where the grid width is the number of points on a row - 1, then the distance between any 2 diagonal grid points is either n+2 or n. This is illustrated in figure 14(a) below where a 4x5 grid point with n = 4-1 = 3 is illustrated. There are two pipelines (line 1 in blue and line 2 in red) that intersect at 2 sections, the first intersection occurring between nodes 5 and 2 in a downward diagonal (n difference), and the second occurring between nodes 7 and 12 in an upward diagonal (n+2 difference).

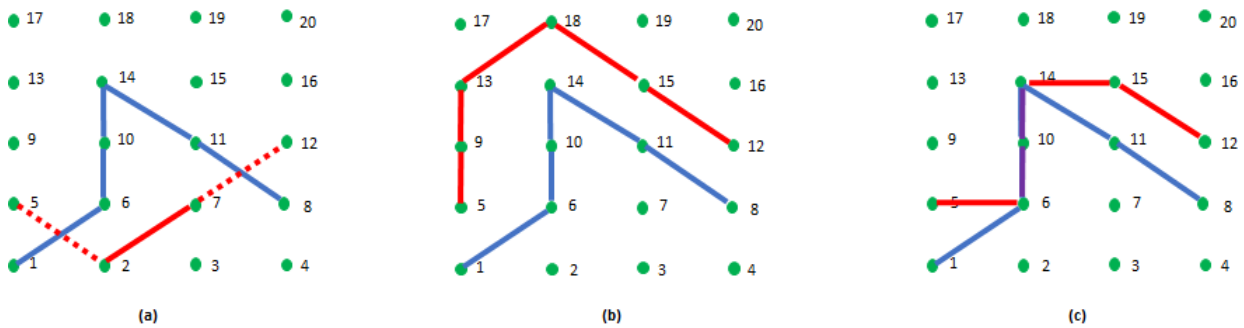



*Figure 14: Illustration showing the effects of the diagonal exclusion zones algorithm. (a) shows paths crossing prior to implementation of diagonal exclusion zones (b) shows one possible realization of path generated after diagonal exclusion zone has been implemented, (c) another possible realization of path generation where 2 paths follow the same nodes in a segment.*

Given these properties, the following algorithm for enforcing diagonal exclusion zones was implemented:

***ALGORITHM FOR DIAGONAL EXCLUSION ZONES***

---

***Input****: G(V, E) a cost surface graph; P(E), a list of graph edges along a generated alternate pipeline; pipeline width n*
***Output****: G'(V, E'), a modified cost surface graph with some edge weights along pipeline vertices set to 1e9*

```
forall edgepair in P:
   if diff(edgepair) = n+2:
      lower_diag = min(edgepair)+1
      upper_diag = max(nodepair)-1
      G.edges[(lower_diag, upper_diag)][weight] = 1e9

   else if diff(edgepair) = n:
      lower_diag = min(nodepair)-1
      upper_diag = max(nodepair)+1
      G.edges[(lower_diag, upper_diag)][weight] = 1e9
   end if
end for
```

---

It becomes important to note that while diagonal edge paths are discouraged by assigning exponentially higher costs, leading to selection of paths that avoid diagonals as shown in figure 14(b), sometimes, pipes may be routed through the same edges as shown in 14(c). Routing through the same edges may be interpreted in one of two ways:

- Pipelines are built side-by-side on the same physical location OR
- Pipelines merge into a larger diameter pipeline and exit as separate streams after some distance.

The actual implementation then becomes a function of operator preferences and technical specifications. The algorithm also presented can be used for existing pipelines to ensure no intersecting path is created across an existing pipeline. Also note that implementing this algorithm each time a new pipeline path is generated is computationally intensive and thus may be switched off if desired.

When scenario (c) occurs, the inlet and outlet points of these merged routes are modeled as transshipment nodes which are utilized in network optimization modeling.

## 3.3 Sequestration Network Optimization Implementation

### 3.3.3 Solver Selection

There are various commercial and open-source optimization solvers that are available to solve linear programming (LP) and mixed-integer programming (MIP) problems. The selection of a solver usually depends on individual preferences, although certain solvers are widely recognized as state-of-the-art in operations research. These solvers include:

- CPLEX, which was developed by IBM and is available for academic use with commercial restrictions.
- Gurobi, which is available as open-source software with commercial limitations.

The SimCCS project utilized CPLEX as the underlying solver for $CO_2$ sequestration optimization. However, running optimization scenarios locally requires complex installation procedures for the ILOG CPLEX Optimization Studio software, which can be avoided by using the SimCCS web version that runs scenarios on pre-installed cloud-based machines.

Gurobi is considered the fastest solver in the world, as demonstrated by benchmark tests for standard optimization tasks, and can be customized to enhance speed using multiple parameters that can be iteratively optimized with grid-search tuning. The Gurobi Application Programming Interface (API) is compatible with various programming languages, including Java, C++, and Python. Since this thesis project is implemented in Python, Gurobi was the preferred choice for implementing the mathematical optimization model. The Gurobi-Python API (gurobipy) can be easily installed using a "pip install" command, and the commands are intuitive, with extensive documentation, training examples, and support available on the Gurobi website.

Both CPLEX and Gurobi solvers are used in operations research for large-scale optimization across multiple industries, including transportation, supply chain, chemical, medical, and others. As such, their software has been commercialized, with licensed versions available for purchase. Free versions of the software are also available but have restrictions on the number of variables and constraints that they can solve. For instance, the CPLEX free version is limited to 1000 variables and 1000 constraints, while the Gurobi free version is limited to 2000 variables and 2000 constraints.

Gurobi's ability to solve problems that are twice the size of CPLEX's free version was another reason for its selection for implementation. If the model becomes too large to solve using the Gurobi free version, a Mathematical Programming System ".mps" file of the mathematical formulation of the problem generated by Gurobi can be saved and passed on to another Python package called PuLP, which uses the completely open-source COIN-OR Branch-and-Cut (CBC) as its default solver. The main drawback of using CBC is that the solve speed is significantly

slower than CPLEX or Gurobi. Therefore, to remove all restrictions on large-scale sequestration optimization projects, it is recommended to apply for a Gurobi academic or professional license.

## 3.4 Solution Visualization

The results of the network optimization can be grouped into two major classes, geospatial and numerical. The geospatial results are inferred indirectly from the selected pipeline arcs to be built connecting the required $CO_2$ sources to sinks and based on the initial cost graph and the shortest path generated, a set of Lat-Long points are available and can be used to generate shape files to be used for external visualization. The numerical results include the volume of $CO_2$ captured, transported, and stored, the lengths of the suggested transport pipelines, and a detailed cost projection.

To visualize the results in an interactive way with high level summaries crucial for operators/researchers, powerful data visualization libraries including Plotly, Mapbox and Matplotlib were leveraged. These packages were used to create dashboard-like summaries that are hosted on the Python web platform called Streamlit. The details of the implementation can be found on the GitHub platform where the software is open-sourced, and in chapter 4, demo cases are used to illustrate and compare/contrast between the tool and SimCCS.

# CHAPTER 4: Results and Discussion

The efforts to develop a solution that can be deployed across multiple user systems and on the web led to the packing of all the code and software into a Streamlit app. This app is called **Sequestrix™** $CO_2$ Network Optimization Software and for the remainder of this document will simply be referred to as Sequestrix.

In this section, a presentation of three demo cases is made, each serving a different purpose. The first demo gives a detailed description of the user interface developed for Sequestrix and highlights a simple example which is used for benchmarking. The benchmarking is done by comparing results with SimCCS which is currently the industry standard. Once the comparison has been made and the results verified, a second comparison is done in demo 2, this time to showcase Sequestrix's ability to handle large scale optimization cases. In the third demo, the ability to include existing pipeline routes as part of the optimization problem is extensively explored.

## 4.1 Demo 1 (Benchmarking) – Proposing Optimization Routes for New Pipelines

### 4.1.1 Problem and Dataset Description

For benchmarking, a simple sequestration problem was formulated as such - There are 3 $CO_2$ emission sources (capture sites) and two storage sites representing saline aquifers for pure sequestration. These sources and sinks are within a 50-mile radius and as such are relatively close by physical distance. The total annual $CO_2$ available for capture is 40 $MTCO_2$/yr. The two sinks are large saline aquifers with a combined $CO_2$ storage capacity of 145 $MTCO_2$. Other details of the sources and sinks are given in Table 3 and 4 below.

The goal of the optimization is as follows:

Given the source capacities, geolocations, and unit costs (capture costs for sources and storage costs for sinks), find:

- Alternate pipeline routes through which $CO_2$ can be routed from the sources to the sinks.
- Calculate the optimal pipeline path to store at least the set target capacity of the duration of the with the minimum cost of the entire project.
- Assess the quality of the results from Sequestrix and make benchmark comparisons to existing SimCCS.

*Table 3: Demo 1 Benchmarking Input Sources*

| ID | UNIQUE NAME | Capture Capacity ($MTCO_2/yr$) | Total Unit Cost ($/tCO_2$) | Lat | Lon |
|---|---|---|---|---|---|
| 1 | Manhattan | 10 | 2 | 35.882 | -97.112 |
| 2 | Germain | 20 | 1.5 | 36.139 | -97.057 |
| 3 | Tbag | 10 | 1.8 | 36.026 | -96.890 |

*Table 4: Demo 1 Benchmarking Input Sinks*

| ID | UNIQUE NAME | Storage Capacity ($MTCO_2$) | Total Unit Cost ($/tCO_2$) | Lat | Lon |
|---|---|---|---|---|---|
| 3 | carlos1 | 80 | -55 | 35.958 | -96.723 |
| 4 | carlos1-2h | 65 | -80 | 36.206 | -96.724 |

### 4.1.2 Introduction to Sequestrix User-Interface and Results

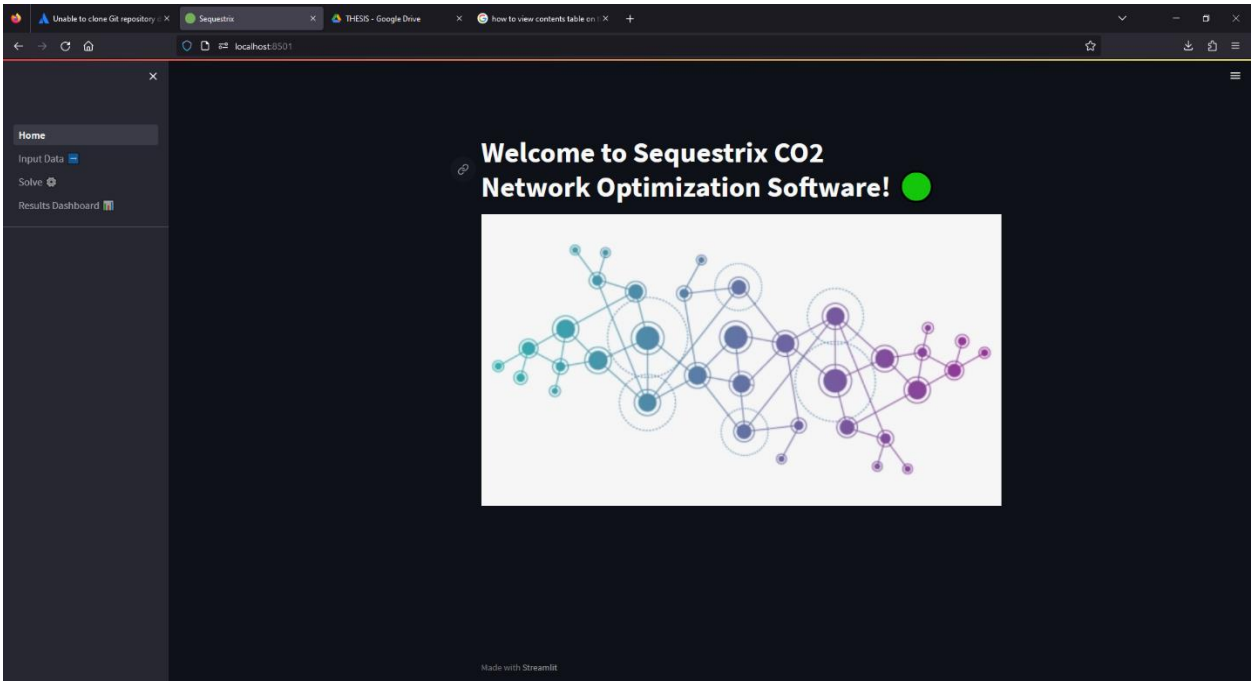



*Figure 15: Landing page of Sequestrix™*

The landing or home page of Sequestrix simply gives a high-level introduction to what the software is about as shown in figure 15. There are three other pages:

- **The Input page:** which can be used to upload csv files with the required information as specified in input tables above. The input page can also be used to import pipeline information in an excel file format and customize specific pipeline settings, however this aspect is further explored in section 4.3 under demo 3. The input page also serves as a dashboard to summarize the data related to the $CO_2$ sources and sinks as is shown in figure 16-18 below:

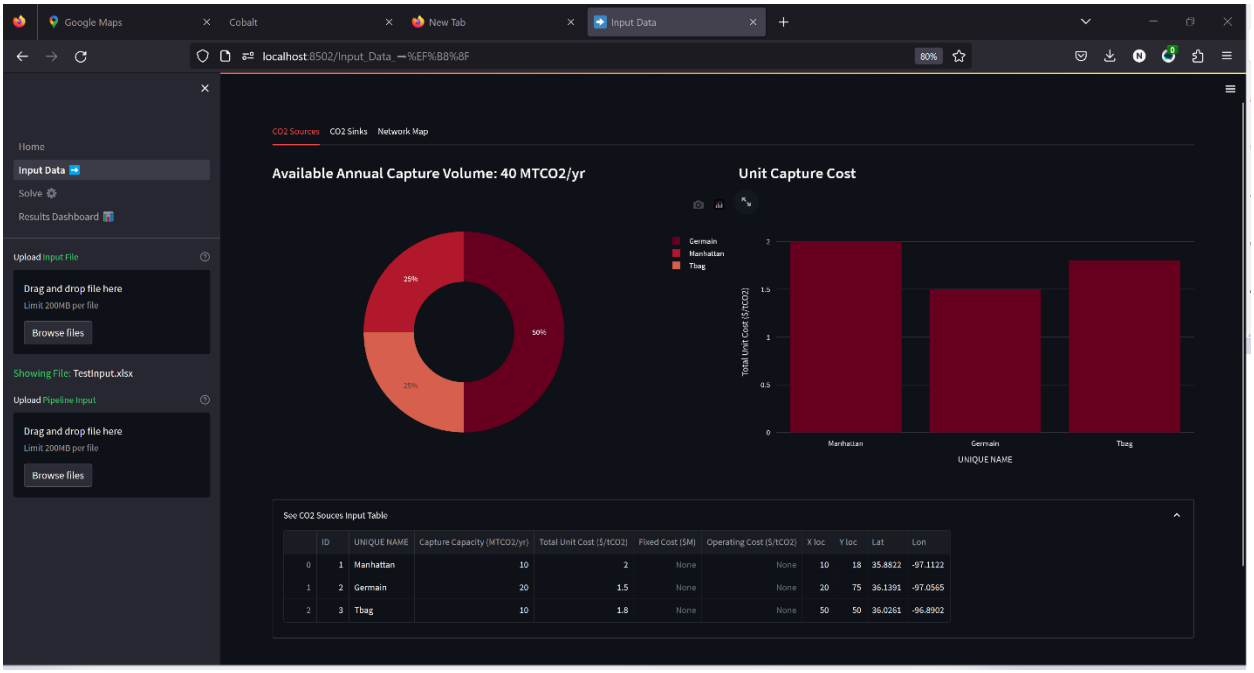



*Figure 16: Sequestrix input data page showing summary dashboard of $CO_2$ sources.*

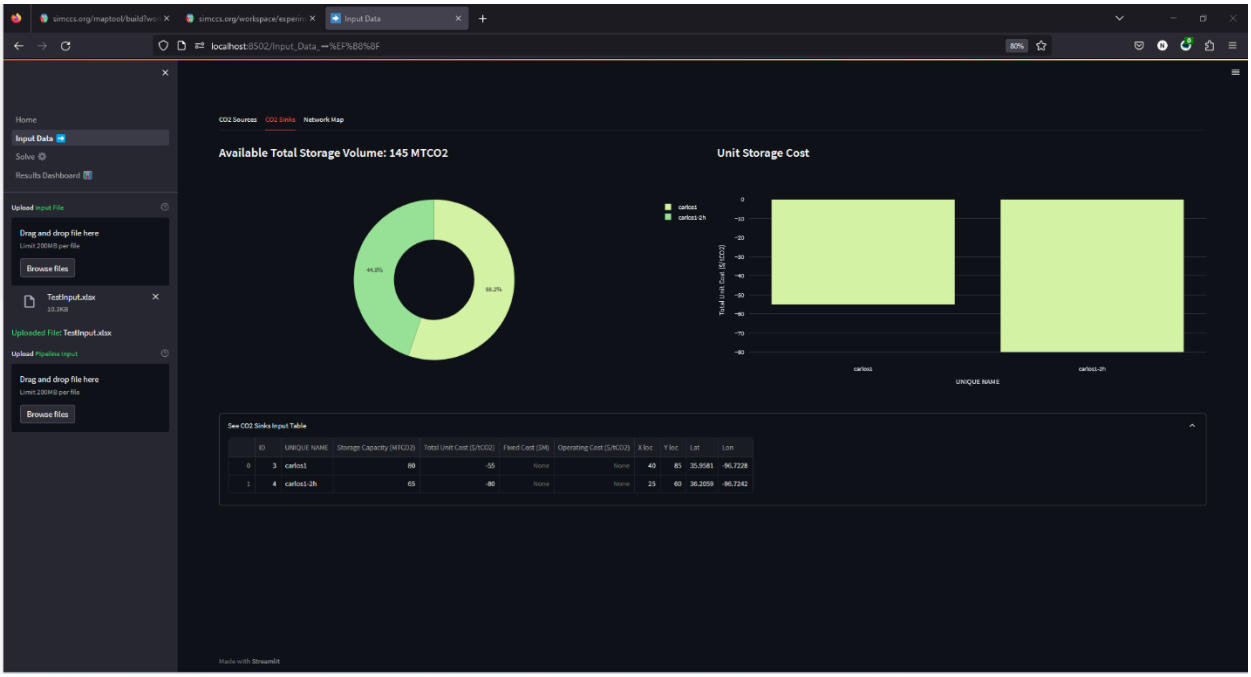



*Figure 17: Sequestrix input data page showing summary dashboard of $CO_2$ sinks.*

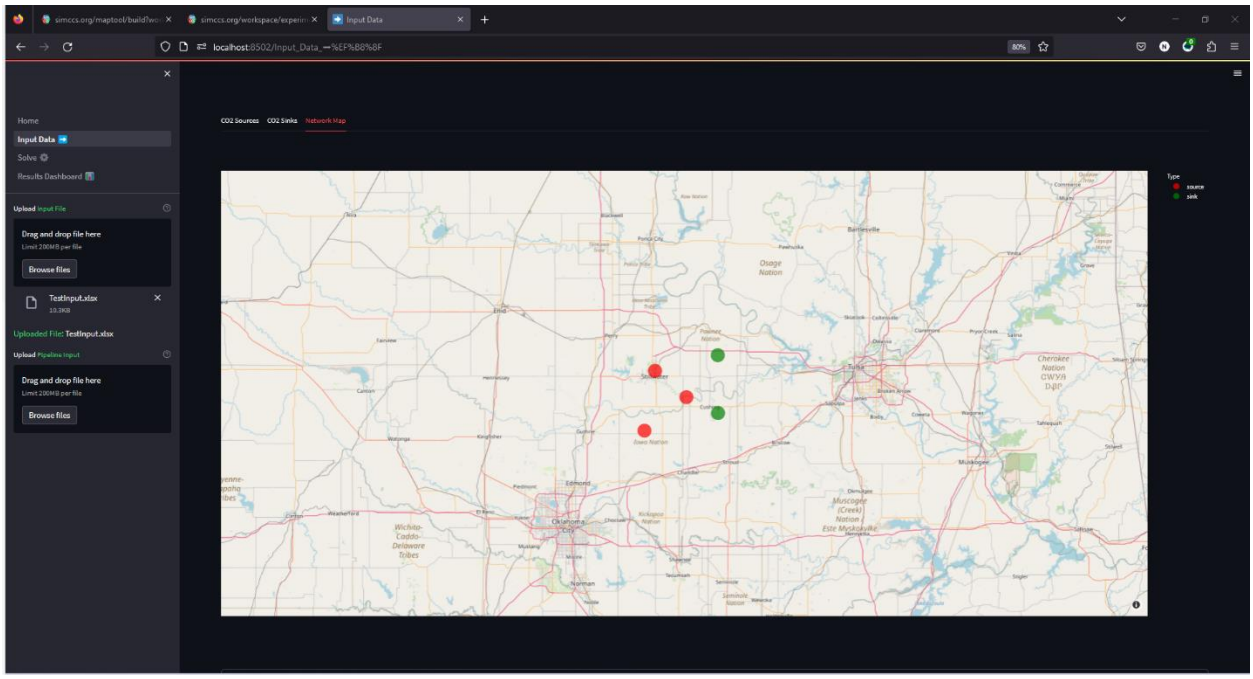



*Figure 18: Sequestrix input data page showing geographic location of sources and sinks in Demo 1 on map.*

- **The Solve Page**: This page can be used to specify three major additional inputs and then run the Network optimization MIP program in the backend. The duration of the project is entered, followed by the minimum $CO_2$ to be sequestered within the specified duration (target) and finally the capital recovery factor. The capital recovery factor (CRF) is defined "*the ratio of a constant annuity to the present value of receiving that annuity for a given length of time"* and is used to determine the present value of a series of equal annual payments for the transport network to be built over the sequestration duration.

  The results of the solve page for demo 1 inputs are the Delaunay triangulation results, the alternate paths generated connecting all sources to sinks within the network, and finally, the optimal pipeline route selected by the optimization engine running locally on the computer. These results are shown in figure 19-21 below:

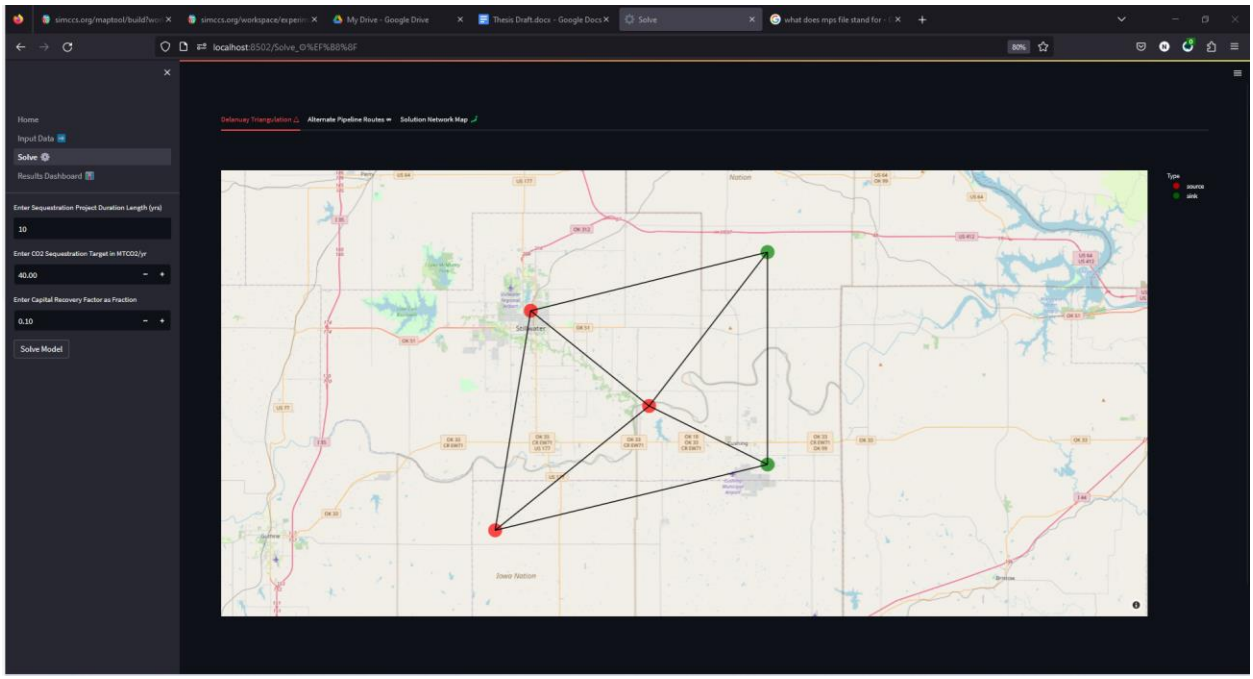



*Figure 19:Sequestrix Solve page showing Delaunay Triangles generated for Demo 1*

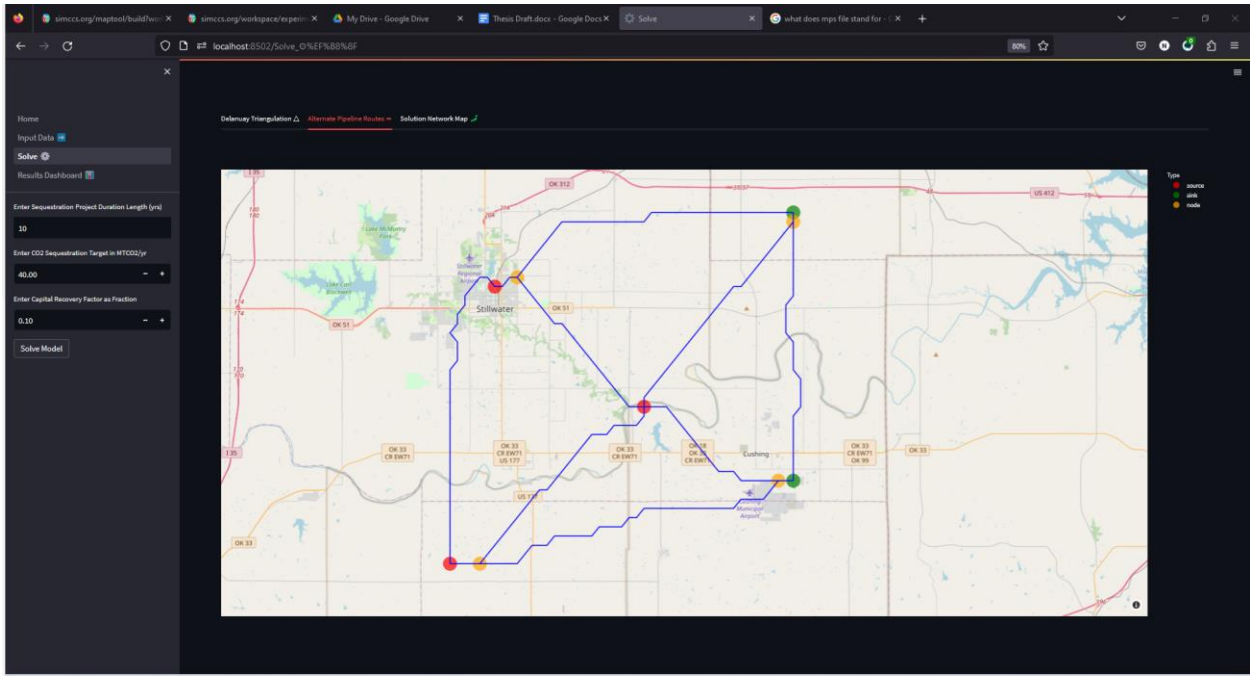



*Figure 20: Sequestrix Solve page showing alternate pipeline network generated for Demo 1 based on Delaunay Triangulation*

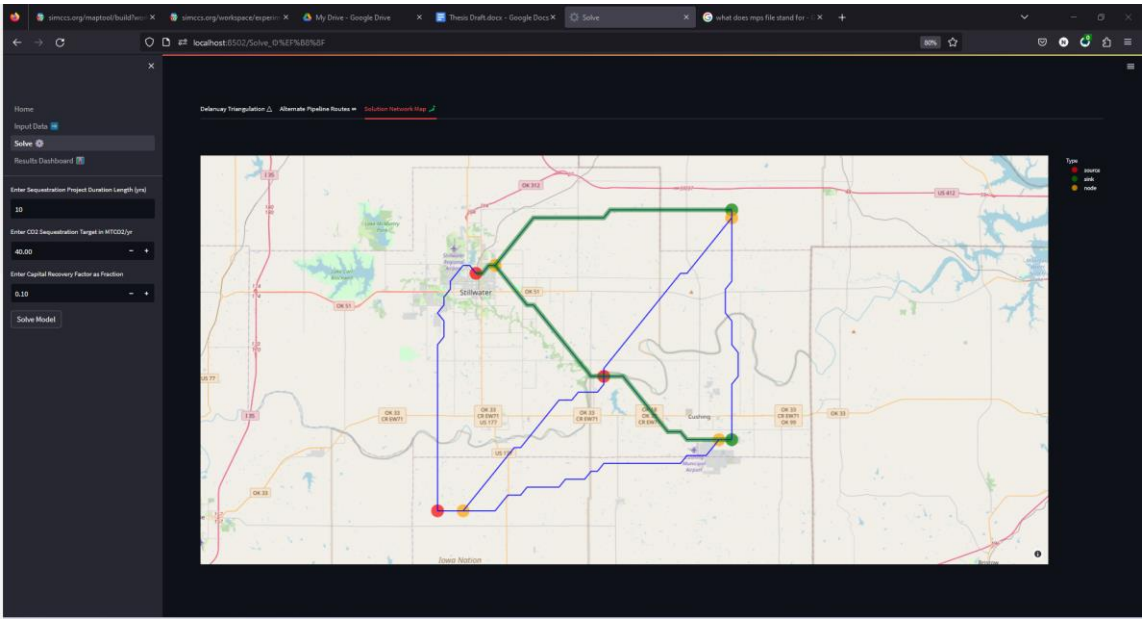



*Figure 21: Sequestrix Solve page with optimal solution path for Demo 1 highlighted in green.*

- **The Results Dashboard Page**: This page was specifically designed to give analytical representation of the key results of network optimization. This is done to highlight key evaluation metrics and to speed up decision making. The results of the optimization are also saved in a csv file which can be inspected for further details.

  The dashboard consists of four sections:

  - ***Overview section*** – which gives key metrics including project duration, total volume of $CO_2$ sequestered, number of sources and sink used and a breakdown of the expected unit costs for sequestration.

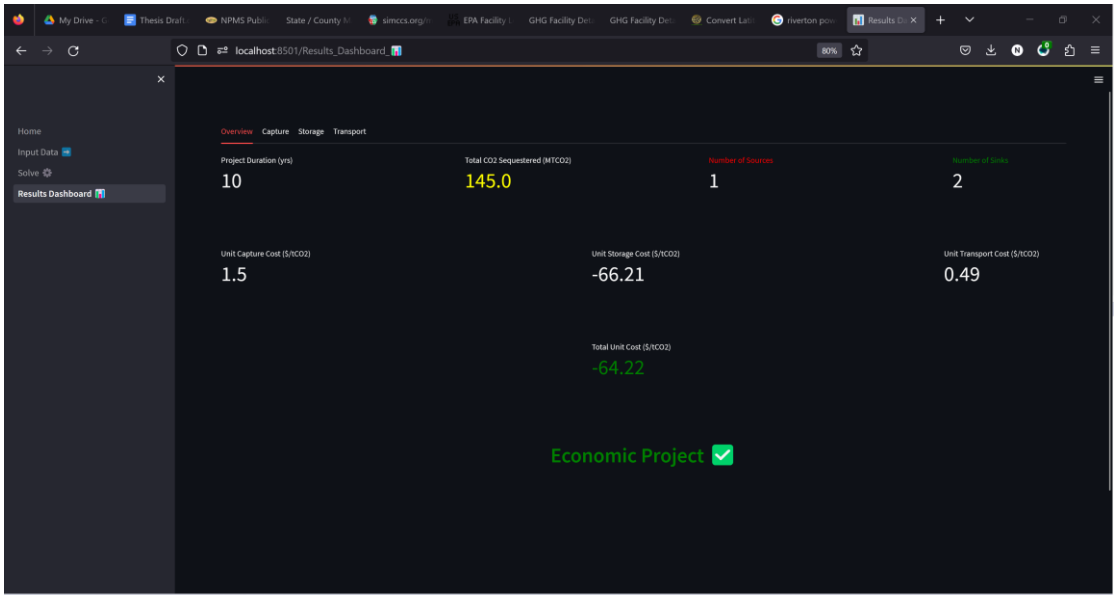



*Figure 22: Sequestrix Results Dashboard page showing key overview results for Demo 1*

- ***Capture section*** – The Capture section in Sequestrix provides a more detailed breakdown of the $CO_2$ capture results, elaborating on the selected sources, total annual capture costs in millions of dollars, capture volumes, and any deviations from the set target. Deviations usually occur if there is a bottleneck in the system. In the provided example, the target capture input into Sequestrix is 40 $MTCO_2$/yr. However, based on the storage capacity of the two sinks provided (145 $MTCO_2$) and the duration of the project (10 years), only a maximum of 14.5 $MTCO_2$/yr can be stored. The 40 $MTCO_2$/yr target was intentionally selected to illustrate the built-in adaptability features of the model. Sequestrix can recognize and adjust to such bottlenecks, providing a more realistic optimization result based on the constraints in the system. This adaptability feature is an advantage when working with

Sequestrix, as it automatically identifies and adjusts for potential bottlenecks in the system, allowing for more accurate and reliable optimization results.

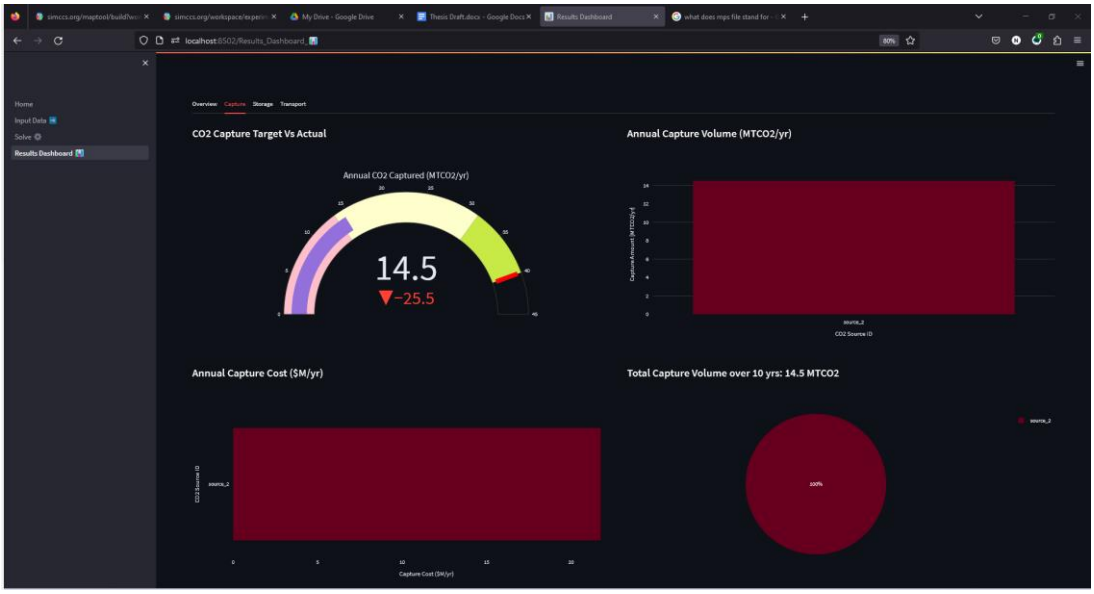



*Figure 23: Sequestrix Results Dashboard page showing $CO_2$ Capture results for Demo 1*

- ***Storage Section*** – gives a detailed breakdown of the storage results, as was done with capture. Plots of costs and storage volumes illustrated.

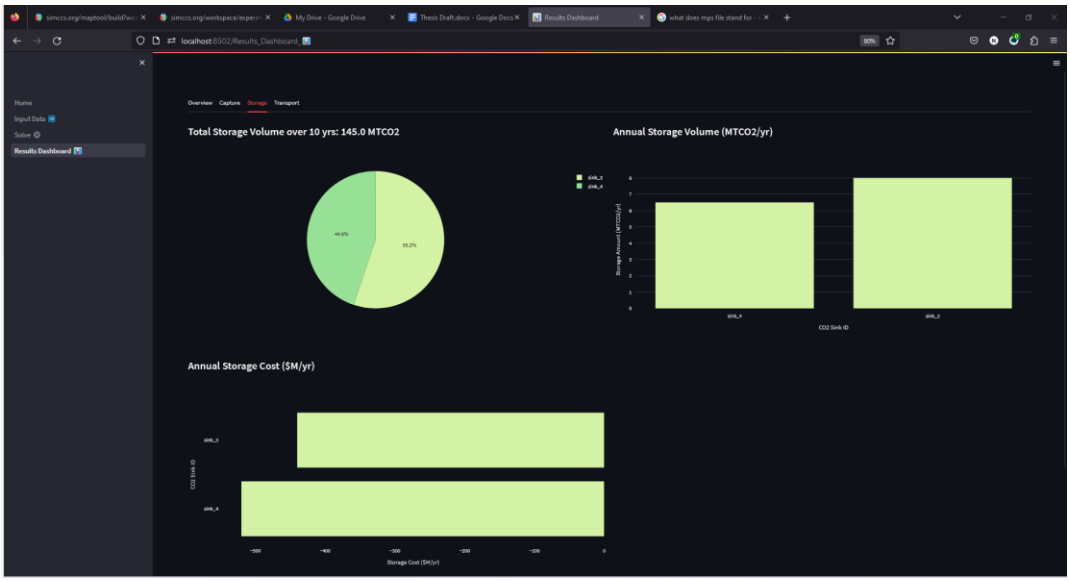



*Figure 24: Sequestrix Results Dashboard page showing $CO_2$ storage results for Demo 1*

- ***Transport section*** – here all the major start and end pipeline points are recorded, this includes sources, sinks, existing or pre-imported pipelines and proposed transshipment nodes along proposed routes. The cost of these pipelines based on calculations detailed in section 3 is also presented.

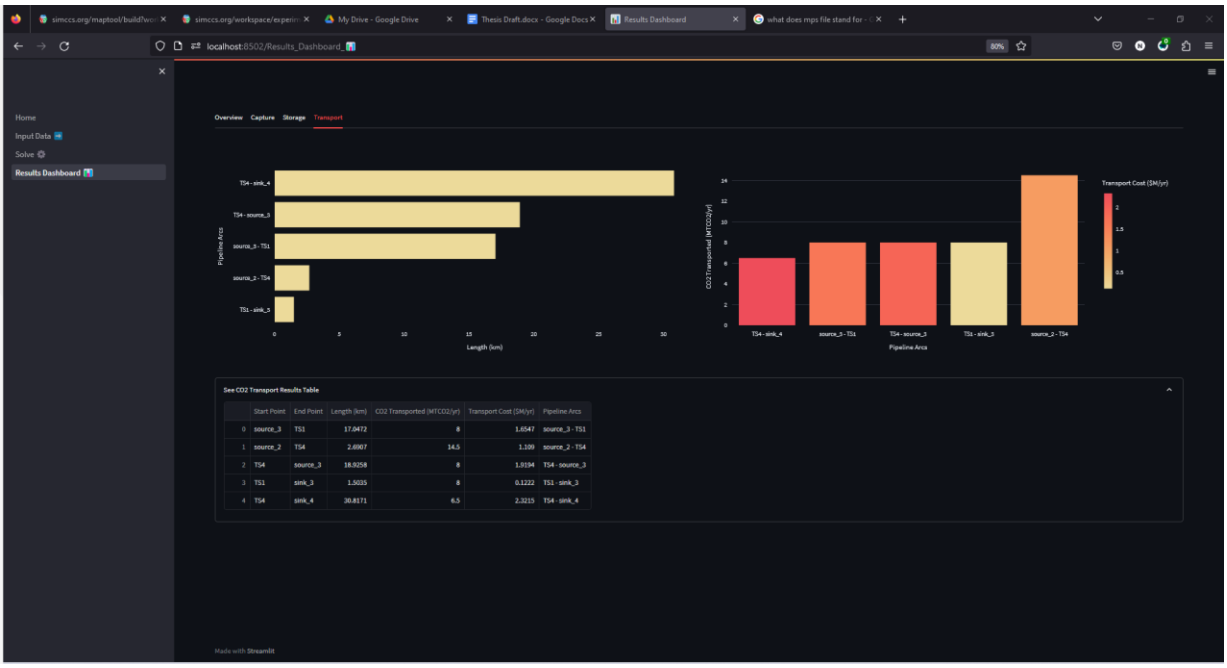



*Figure 25: Sequestrix Results Dashboard page showing transport pipeline result details for Demo 1*

### 4.1.3 SimCCS Interface and Results

To conduct an unbiased comparison, a local instance of SimCCS was run on a personal computer (an MSI GE Raider 66, with 32GB RAM, 11th gen core i9 3.0Ghz processor). A similar CSV input file was generated for SimCCS, which can be uploaded to the tool, and the three main inputs (duration, CRF, and target) were entered before running the model.

Upon completion of the optimization, the suggested optimal transport route connecting the sources and sinks can be displayed, and a high-level summary of the unit and annual costs are shown on

the tool. A more detailed breakdown of the pipeline connections is exported to a solution CSV file, which can be inspected for further details.

It is important to note that the local version of SimCCS does not have a feature that allows it to check for bottlenecks during optimization and adjust capture or storage properties. The user must inspect the input data to ensure the optimization will work before solving with CPLEX. This limitation requires users to be more cautious when preparing their input data and setting up the optimization problem, as opposed to the automated bottleneck checking and adjustments provided in Sequestrix. Figure 26 shows the SimCCS interface and results.

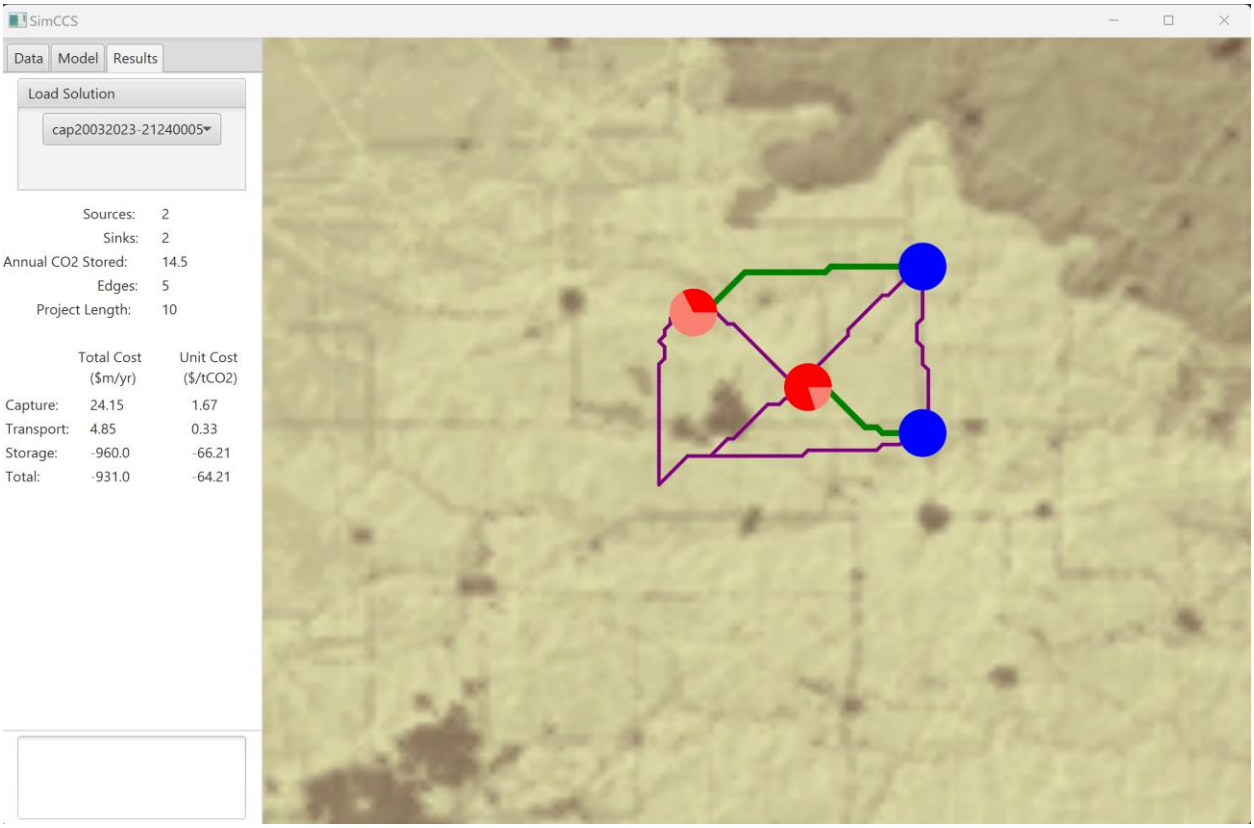



*Figure 26: SimCCS user interface showing results summary for Demo 1*

### 4.1.4 Sequestrix vs SimCCS Detailed Benchmarking

To conduct proper benchmarking, three key metrics were defined to compare the solutions. These metrics include:

1. Speed(Runtime): This includes the time taken to generate alternate pipeline routes and the time taken to solve the optimization problem.
2. Unit Costs: These include capture, storage, and transport costs.
3. Pipeline Lengths: The total length of the pipelines in the solutions.

In addition to these metrics, a qualitative evaluation was also conducted, examining the generated alternate pipeline routes and the selection of source and sink pairings. This qualitative assessment provided insights into the differences in the solutions proposed by the two platforms, Sequestrix and SimCCS, and helped to further evaluate their performance and effectiveness in solving $CO_2$ sequestration network optimization problems.

***Quantitative Metrics***

*Table 5: Comparison of SimCCS and Sequestrix results for Demo 1*

| Metric | SimCCS (local) | Sequestrix |
|---|---|---|
| **Unit Capture Cost ($/ton $CO_2$)** | 1.67 | 1.50 |
| **Unit Transport Cost ($/ton $CO_2$)** | 0.33 | 0.49 |
| **Unit Storage Cost ($/ton $CO_2$)** | -66.21 | -66.21 |
| **Unit Total Cost ($/ton $CO_2$)** | -64.21 | -64.22 |
| **Runtime** | 600 | 190 |
| **Total Pipeline Length (km)** | 51.99 | 70.98 |

Analysis of table 5 above reveals that solutions generated by Sequestrix and SimCCS are comparable but slightly different. The sources chosen for capture in SimCCS are Germain and Tbag, each source providing 6.5 $MTCO_2$/yr and 8 $MTCO_2$/yr, respectively. On the other hand, Sequestrix selects generates a solution where the entire 14.5$MTCO_2$/yr is taken from Germain which is the cheapest source of $CO_2$ capture, and as such, the overall unit capture cost is lower. This is flipped when considering transportation costs, as in the SimCCS solution, two shorter length pipelines are built from Germain to carlos1-2h sink and from Tbag to carlos1 sink, while 2 pipelines are built from Germain to both sinks in the Sequestrix solution. Overall, the reduction in capture cost achieved by Sequestrix is balanced out by the increased transport cost, however, when considering the overall unit total cost, Sequestrix slightly edges out the solution generated by SimCCS.

A MIP objective function may have several non-unique optimal solutions which lie along a pareto plane. Each solution that lies on this pareto plane has a different combination of the decision variable solutions which generate comparable results. The number of optimal solutions generated also depends on the MIP gap set during optimization, which denotes an acceptable deviation from the most optimal solution. Both SimCCS and Sequestrix do not have a set MIP gap and hence CPLEX and Gurobi use default MIP gap values in generating solutions.

There is also a 68% speed improvement in running SimCCS vs Sequestrix on the specified local PC and this becomes useful especially when planning for multiple scenarios.

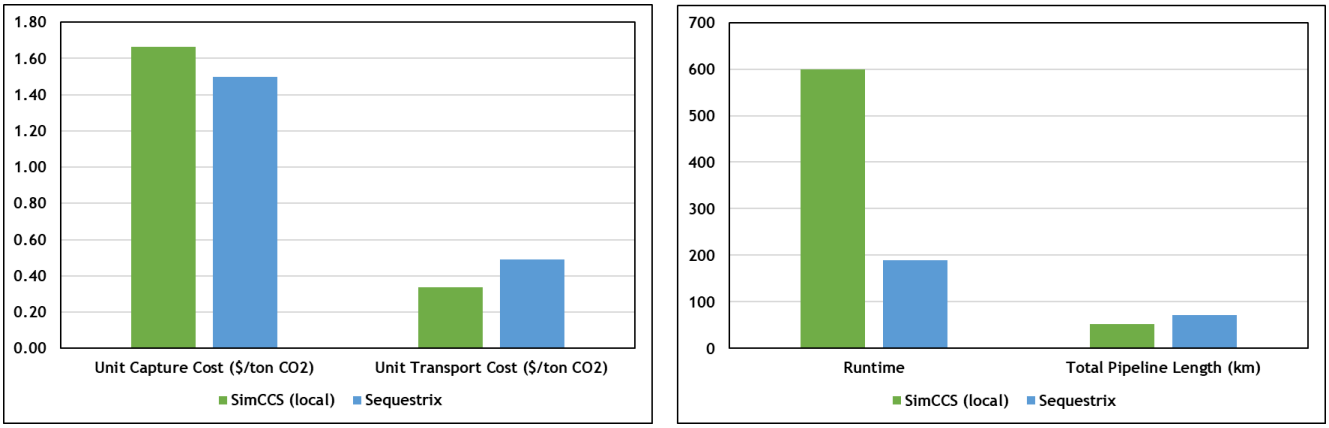



*Figure 27: Comparison Plot of SimCCS vs Sequestrix results for Demo 1*

### *Qualitative Metrics*

A more detailed examination of the alternate pipeline routes generated by both tools reveal slight differences in the paths. Ideally since the paths generated rely on Dijkstra's shortest path algorithm in both tools, and the cost surface graph is also the same for both tools, one would expect that the selected shortest paths should converge, but this may not necessarily happen as illustrated in figure 28(b), where the circle highlights the path connecting Manhattan Source to carlos1 sink is visibly different. These differences can be attributed to two reasons explained briefly below:

- Non-unique shortest paths – the Dijkstra's algorithm is a greedy algorithm which aims to return the first shortest path it finds, and while the length of this path is guaranteed to the shortest, there may be multiple other paths with the same path length.
- Pre-defined post-processing pipeline algorithms – Because multiple shortest paths are generated between different source sink pairs because of the Delaunay

triangulation, sometimes the paths generated between two different node pairs may fall along the same route except for a few nodes on the graph, these odd-looking paths generated may require post processing before the MIP optimization is carried out. The difference in post-processing algorithms may lead to different final paths.

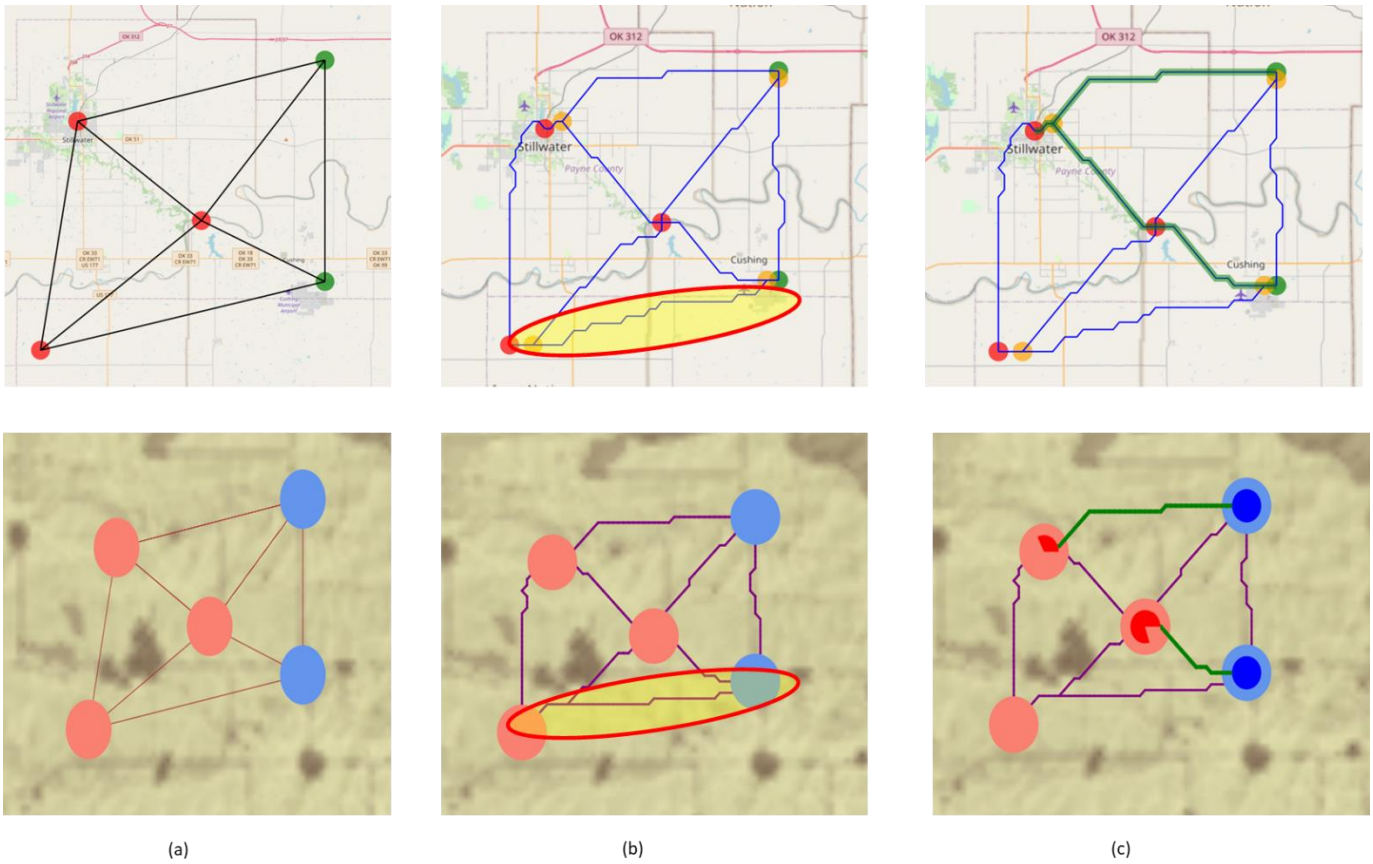



*Figure 28: Detailed graphical comparison of solutions generated by SimCCS and Sequestrix for Demo 1. Plots for Sequestrix are located above and SimCCS below. (a) shows the results of Delaunay triangulation which look identical for both tools, (b) shows the alternate (or candidate) networks generated by both tools. The yellow oval line highlights differences in LCP generated, (c) shows the resulting optimal network selected after optimization, different paths are selected for both tools.*

In summary, demo 1 has illustrated that Sequestrix is at least as good as SimCCS in solving the small-scale $CO_2$ Network optimization problem and suggesting optimal transport routes. This comes with the added benefits of better visual representation of results and faster runtimes.

## 4.2 **Demo 2 (Scalability) – Solving Large Scale Problems Across Oklahoma**

In Demo 1, the key thing that was highlighted is the ability of Sequestrix to formulate and solve the $CO_2$ network optimization problem. As such, the geolocation of sources and sinks and all associated costs were arbitrarily assigned and had no scientific basis. Having satisfactorily demonstrated Sequestrix' ability, more care is taken in demo 2 to assess a research problem with real sources, sinks, geolocations, and associated costs.

Demo 2 tests Sequestrix on a problem previously solved and published in a self-authored SPE paper (Jamal et al, 2021). The geographical scope of the problem covers the entire Oklahoma state and allows a demonstration of scalability in usage of Sequestrix.

### 4.2.1 Problem Description

Oklahoma is recognized for its abundant $CO_2$ sources, pipelines, and reservoirs where oil and gas companies have been employing $CO_2$ injection into geological structures for enhanced oil recovery (EOR) for several years. We employed Sequestrix and SimCCS, software tools that combines economic and engineering aspects, to consolidate infrastructure concerning $CO_2$ sources, pipelines, and geological formations. The IRS-endorsed tax incentive initiative, 45Q, has encouraged many oil and gas companies to contribute to $CO_2$ reduction and global warming mitigation by capturing $CO_2$ from a variety of sources, identifying optimal pipeline routes, and selecting the most secure locations for EOR-based injections or deep saline aquifers for sequestration.

### 4.2.2 Costs – Capture, Transport and Storage

***Capture costs.***

For this study, capture costs were obtained from ranges published by The Great Plains Institute in as discussed in section 2.

***Transport costs***

The transportation costs utilized for this study are the same as generated using the linear trendlines described in section 2.4.5 which are based on the amount of $CO_2$ flow in pipeline.

***Storage costs***

Estimating storage costs relies on the storage type, with $CO_2$ potentially stored in oil and gas reservoirs, saline aquifers, coal bed seams, deep oceans, or via mineral carbonization. Oklahoma, a terrestrial region without surrounding oceans, considers only geological storage. Storage costs vary depending on whether $CO_2$ is used for enhanced oil recovery (EOR) or simply for storage. Injection costs depend on location and depth, with shallow onshore wells having the lowest costs and deep offshore wells the highest. Injection costs are estimated to range between \$0.3-\$8 USD per ton of $CO_2$ stored. The 2021 45Q tax credit of \$35/t$CO_2$ stored in hydrocarbon reservoirs is applied, but the revenue from additional hydrocarbon production is not represented due to data unavailability. For the Oklahoma study, a pessimistic value of -\$31/t$CO_2$ is used as the storage cost for further evaluation.

### 4.2.3 $CO_2$ Emission sources

***$CO_2$ sources***

The U.S. Environmental Protection Agency (EPA) monitors greenhouse gas emissions and their sources, storing the information in their Facility Level Information on Green House gases (FLIGHT) tool. As at the time of this study in 2021, the FLIGHT tool warehoused GHG emissions data from 2010 to 2019 for individual states, categorized into nine major sectors. In 2019, Oklahoma had 151 total emission sources, with contributing facilities ranked across these sectors, as depicted in figure 29.

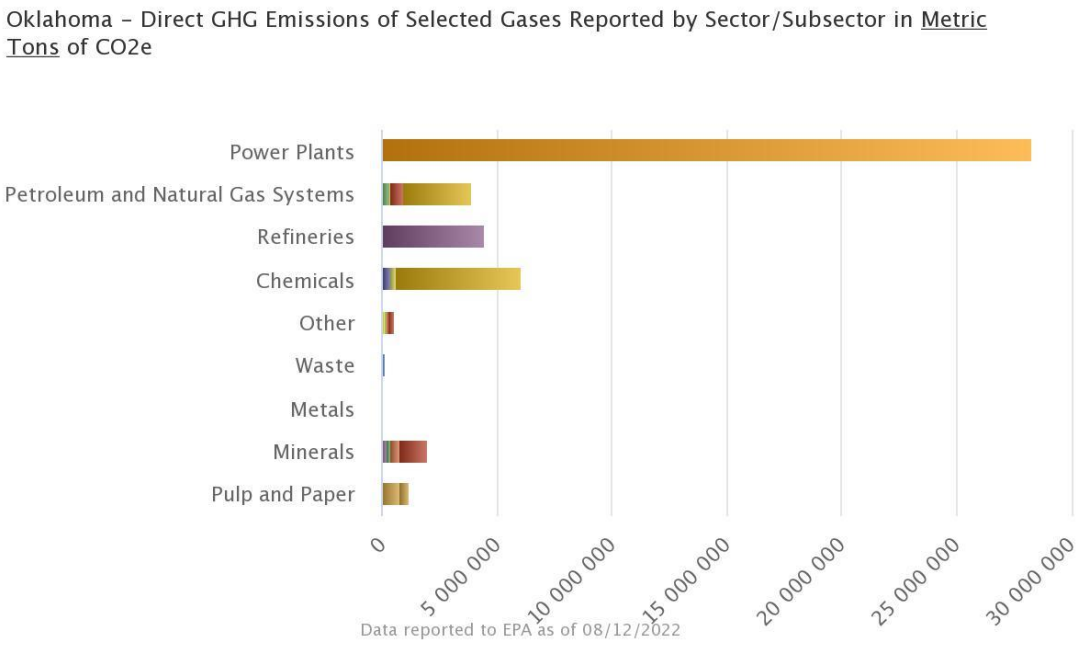



*Figure 29: 2019 GHG emissions in Oklahoma by sector (Source: EPA FLIGHT Tool)*

The power sector emerged as the primary emission source in Oklahoma, accounting for over 60% of GHG emissions. It is followed by the chemical, refining, petroleum and natural gas processing,

and minerals industries. This study utilizes the average annual emission per plant from 2016 to 2019 to represent $CO_2$ emissions in megatons (MT) of $CO_2$ per year. Figure 30 displays the distribution of these plants across various counties and demographic areas in Oklahoma.

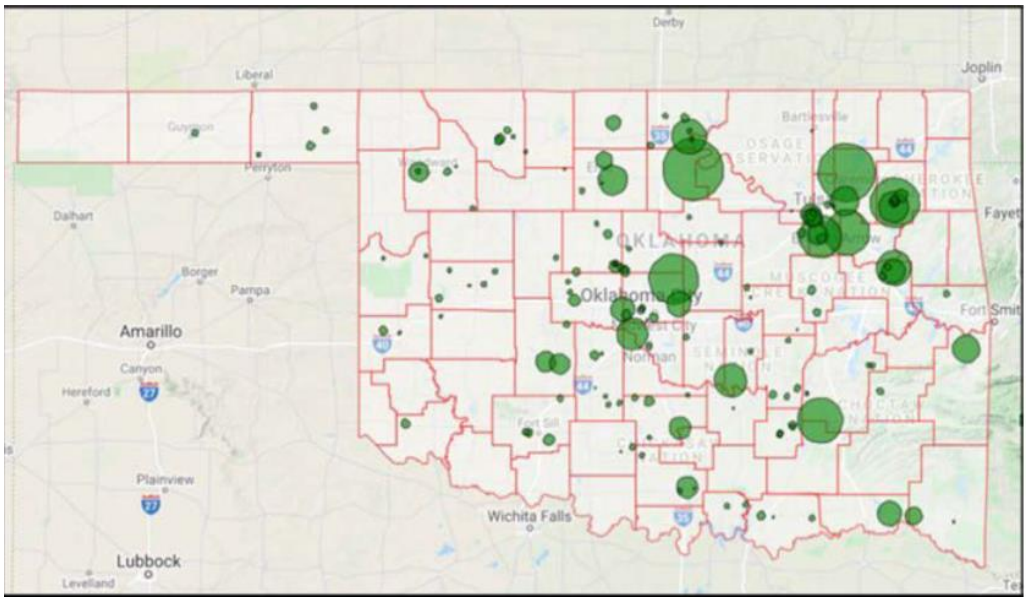



*Figure 30: Map of Oklahoma Showing $CO_2$ emissions from sources across counties, bubble size represents emission volume (DaneshFar et al., 2021)*

***Streamlining $CO_2$ source selection***

To further refine the selection of $CO_2$ emission sources in Oklahoma, the sources were assessed based on the 2018 45Q federal tax credit system requirements. This screening process identified the top 36 $CO_2$ emission sources eligible for 45Q tax credits due to their emission capacity.

*Table 6: Demo 2 top 36 Sources Obtained after application of 45Q eligibility screening*

| DEMO 2 SOURCES | | | | | |
|---|---|---|---|---|---|
| **ID** | **UNIQUE NAME** | **Capture Capacity (MTCO2/yr)** | **Total Unit Cost ($/tCO2)** | **Lat** | **Lon** |
| 1 | Chisholm Plant | 0.1 | 14 | 35.775 | -97.741 |
| 2 | Cana Gas Plant | 0.1 | 14 | 35.535 | -98.099 |
| 3 | OHL NGLP Medford Plant | 0.1 | 14 | 36.776 | -97.756 |
| 4 | KOCH FERITLIZER ENID LLC ENID NITROGEN PLT | 0.4 | 17 | 36.379 | -97.761 |
| 5 | VERDIGRIS PLT | 0.4 | 17 | 36.234 | -95.719 |
| 6 | Redbud Power Plant | 2.3 | 75 | 35.685 | -97.224 |
| 7 | Sooner | 3.3 | 56 | 36.454 | -97.053 |
| 8 | Chouteau Power Plant | 2.2 | 75 | 36.221 | -95.276 |
| 9 | Northeastern | 3 | 75 | 36.432 | -95.701 |
| 10 | Tenaska Kiamichi Generating Station | 1.9 | 75 | 34.683 | -95.935 |
| 11 | TERRA INTERNATIONAL (OKLAHOMA) INC | 0.2 | 17 | 36.437 | -99.471 |
| 12 | Green Country Energy, LLC | 1.6 | 75 | 35.983 | -95.935 |
| 13 | McClain Energy Facility | 0.9 | 75 | 35.298 | -97.590 |
| 14 | PRYOR CHEMICAL COMPANY | 0.1 | 17 | 36.241 | -95.278 |
| 15 | Oneta Energy Center | 1.8 | 75 | 36.012 | -95.697 |
| 16 | Grand River Dam Authority | 0.8 | 75 | 36.191 | -95.289 |
| 17 | Seminole (2956) | 1 | 56 | 34.968 | -96.724 |
| 18 | Muskogee | 1.1 | 75 | 35.762 | -95.285 |
| 19 | Hugo | 0.6 | 56 | 34.016 | -95.321 |
| 20 | Horseshoe Lake | 0.6 | 75 | 35.509 | -97.179 |
| 21 | River Valley Generating Station | 0.7 | 56 | 35.193 | -94.647 |
| 22 | Mustang | 0.5 | 75 | 35.471 | -97.673 |
| 23 | EAGLE MATERIALS, INC. | 0.3 | 56 | 36.194 | -95.812 |
| 24 | CONTINENTAL CARBON Ponca City Plant | 0.2 | 30 | 36.666 | -97.072 |
| 25 | HOLCIM INCORPORATED | 0.4 | 56 | 34.768 | -96.697 |
| 26 | Phillips 66 Ponca City Refinery | 1.7 | 56 | 36.682 | -97.090 |
| 27 | LONE STAR IND INC DBA BUZZI UNICEM USA PRYOR CEMENT PLANT | 0.3 | 56 | 36.272 | -95.223 |
| 28 | US LIME COMPANY-ST. CLAIR | 0.1 | 56 | 35.582 | -94.819 |
| 29 | VALERO REFINING CO -OKLAHOMA VALERO ARDMORE REFINERY | 0.9 | 56 | 34.206 | -97.104 |
| 30 | OXBOW CALCINING LLC | 0.3 | 56 | 36.518 | -97.839 |
| 31 | WYNNEWOOD REFINING CO | 0.7 | 56 | 34.629 | -97.169 |
| 32 | HOLLYFRONTIER TULSA REFINING LLC - EAST | 0.5 | 56 | 36.118 | -96.001 |
| 33 | HOLLYFRONTIER TULSA REFINING LLC - WEST | 0.4 | 56 | 36.140 | -96.015 |
| 34 | Covanta WBH | 0.1 | 39 | 36.132 | -96.017 |
| 35 | GP MUSKOGEE MILL | 0.5 | 39 | 35.740 | -95.287 |
| 36 | International Paper - Valliant Mill | 0.3 | 39 | 33.998 | -95.112 |

### 4.2.4 $CO_2$ Storage

#### *Storage Assets and Sites*

$CO_2$ sink candidates in Oklahoma were pinpointed as injection wells. The Oklahoma Corporation Commission collects underground injection data, including well names, operators, geolocations, and annual injection volumes, making it accessible to the public via their online repository. The database contained yearly injection volumes and data from 2011 to 2019 were selected.

### *Sink Clusters*

An effort was made to aggregate $CO_2$ injection sites, or sinks, into clusters to reduce redundancy and computational time for the subsequent source-sink matching discussed in later sections. To create these clusters, a machine learning clustering algorithm called 'K-MEANS clustering' (Lloyd, 1957) was utilized. This algorithm grouped injection sites into clusters using Euclidean distances and calculated a centroid to represent them. The well names and geographic locations were the only information used to generate these clusters. The implementation of this algorithm resulted in seven clusters belonging to six operators and 14 independent injection sites that fell under the same six operators and an additional two operators. This effectively reduced 245 injection sites to 21.

The average $CO_2$ injection rate per well per operator was estimated by averaging the annual injection volumes between 2011 and 2018 and dividing by the maximum number of wells within those years. This generated a pessimistic value with significant potential upsides. The value was aggregated for the clusters containing more than one well.

*Table 7: Demo 2 Sink clusters obtained after application of K-MEANS to point injection wells*

| DEMO 2 SINKS | | | | | |
|---|---|---|---|---|---|
| ID | UNIQUE NAME | Storage Capacity (MTCO2) | Total Unit Cost ($/tCO2) | Lat | Lon |
| 1 | Cluster 1 | 4.01 | -31 | 36.747 | -101.100 |
| 2 | Cluster 2 | 3.43 | -31 | 34.790 | -97.615 |
| 3 | Cluster 3 | 20.99 | -31 | 36.876 | -101.630 |
| 4 | Cluster 4 | 1.8 | -31 | 35.300 | -98.295 |
| 5 | Cluster 5 | 4.18 | -31 | 36.515 | -100.912 |
| 6 | Cluster 6 | 3.24 | -31 | 34.473 | -97.441 |
| 7 | Cluster 7 | 2.49 | -31 | 34.421 | -97.614 |
| 8 | i1 | 0.11 | -31 | 35.182 | -98.201 |
| 9 | i2 | 0.11 | -31 | 35.184 | -98.201 |
| 10 | i3 | 0.11 | -31 | 35.189 | -98.201 |
| 11 | i4 | 0.31 | -31 | 36.886 | -101.511 |
| 12 | i5 | 0.31 | -31 | 36.876 | -101.800 |
| 13 | i6 | 0.31 | -31 | 36.891 | -101.013 |
| 14 | i7 | 0.31 | -31 | 36.843 | -101.506 |
| 15 | i8 | 0.31 | -31 | 36.856 | -101.211 |
| 16 | i9 | 0.31 | -31 | 36.567 | -101.664 |
| 17 | i10 | 0.12 | -31 | 34.482 | -97.707 |
| 18 | i11 | 0.12 | -31 | 34.506 | -97.598 |
| 19 | i12 | 0.12 | -31 | 35.045 | -97.852 |
| 20 | i13 | 0.12 | -31 | 35.060 | -97.738 |
| 21 | i14 | 0.01 | -31 | 34.932 | -98.148 |

### 4.2.5 $CO_2$ Network Optimization Modeling Results

The annual sequestration target was 2.14 $MTCO_2$/yr for a duration of 20 years. This target represents the total storage capacity of all sinks divided by the sequestration duration. A capital recovery factor of 10% was selected for this run and the results for both Sequestrix and SimCCS are presented in table 8 with pipeline routes shown in figure 31 and 32.

*Table 8: Comparison of SimCCS and Sequestrix results for Demo 2*

| Metric | SimCCS (local) | Sequestrix |
|---|---|---|
| **Unit Capture Cost ($/ton $CO_2$)** | 23.87 | 23.91 |
| **Unit Transport Cost ($/ton $CO_2$)** | 21.03 | 19.48 |
| **Unit Storage Cost ($/ton $CO_2$)** | -31.00 | -31.00 |
| **Unit Total Cost ($/ton $CO_2$)** | 13.90 | 12.39 |
| **Runtime (seconds)** | 1800 | 280 |
| **Total Pipeline Length (km)** | 1108.98 | 1206.58 |

In the results generated by Sequestrix, the unit capture cost is higher than in SimCCS, but the transport cost is significantly lower. Consequently, the overall total cost of the solution produced by Sequestrix is $1.51 cheaper than that generated by SimCCS, although it necessitates a longer pipeline length. A detailed analysis of the candidate and solution networks generated by these two solutions provides insight into the differences in pipeline length.

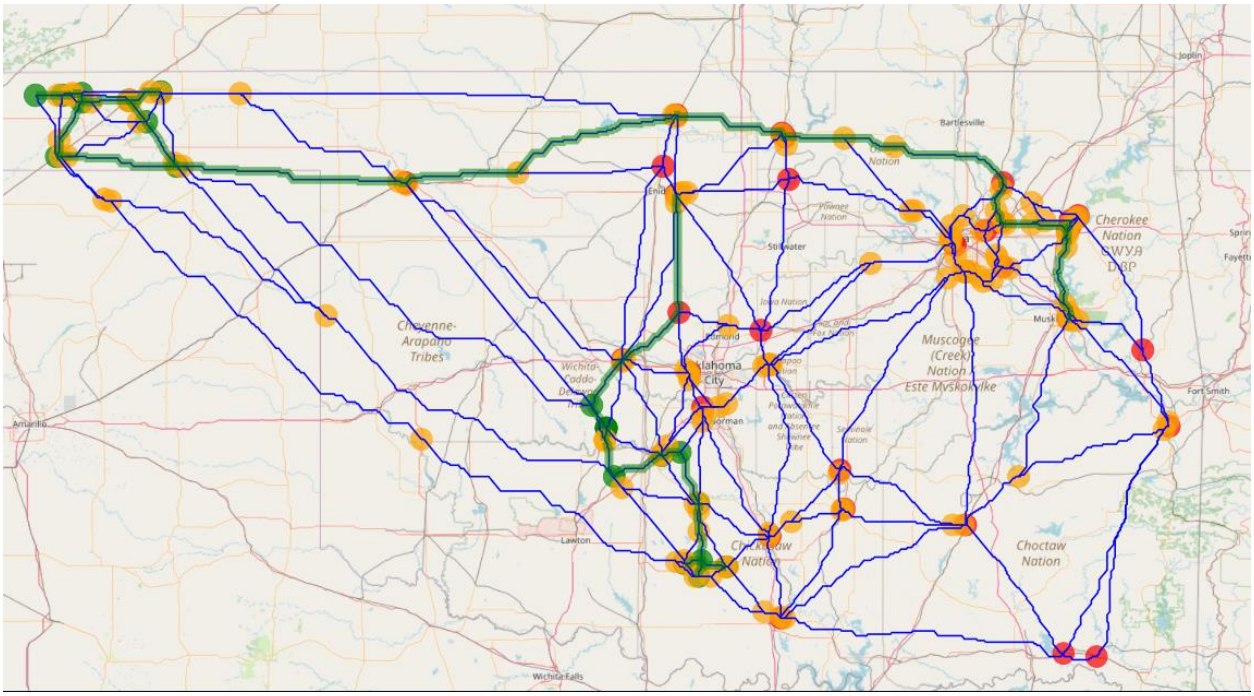



*Figure 31: Sequestrix result view on map surface, the green circles represent $CO_2$ sinks, red represents $CO_2$ sources and yellow are transshipment nodes. The green path highlighted is the optimal pipeline network while the other blue lines represent alternate pipeline networks.*

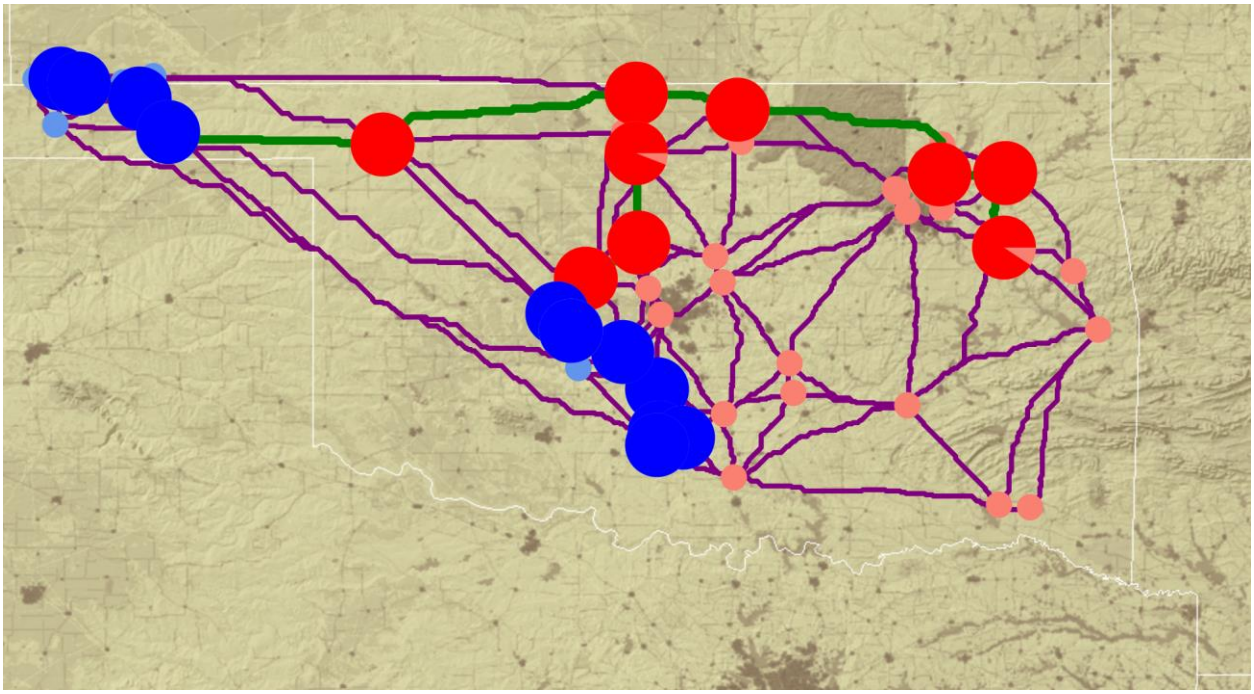


*Figure 32: SimCCS results for Demo 2. The red circles represent $CO_2$ sources and blue represents $CO_2$ sinks. Circles that are highlighted are the selected optimal assets and the green path shows the optimal pipeline network while the other purple lines represent the candidate network.*

In Figure 33(a) and 33(b), it is evident that the selected sinks in the mid-southern section of Oklahoma are not the same, resulting in varying pipeline routes. In Figure 34(a) and 34(b), we observe that in the mid-north to northwest regions, the pipeline routes differ between SimCCS and Sequestrix. Although the paths in Sequestrix results are longer, the construction costs are lower, highlighting the significance of generating alternate pipeline routes, as well as other factors such as post-processing algorithms, MIP gap, and solver selection.

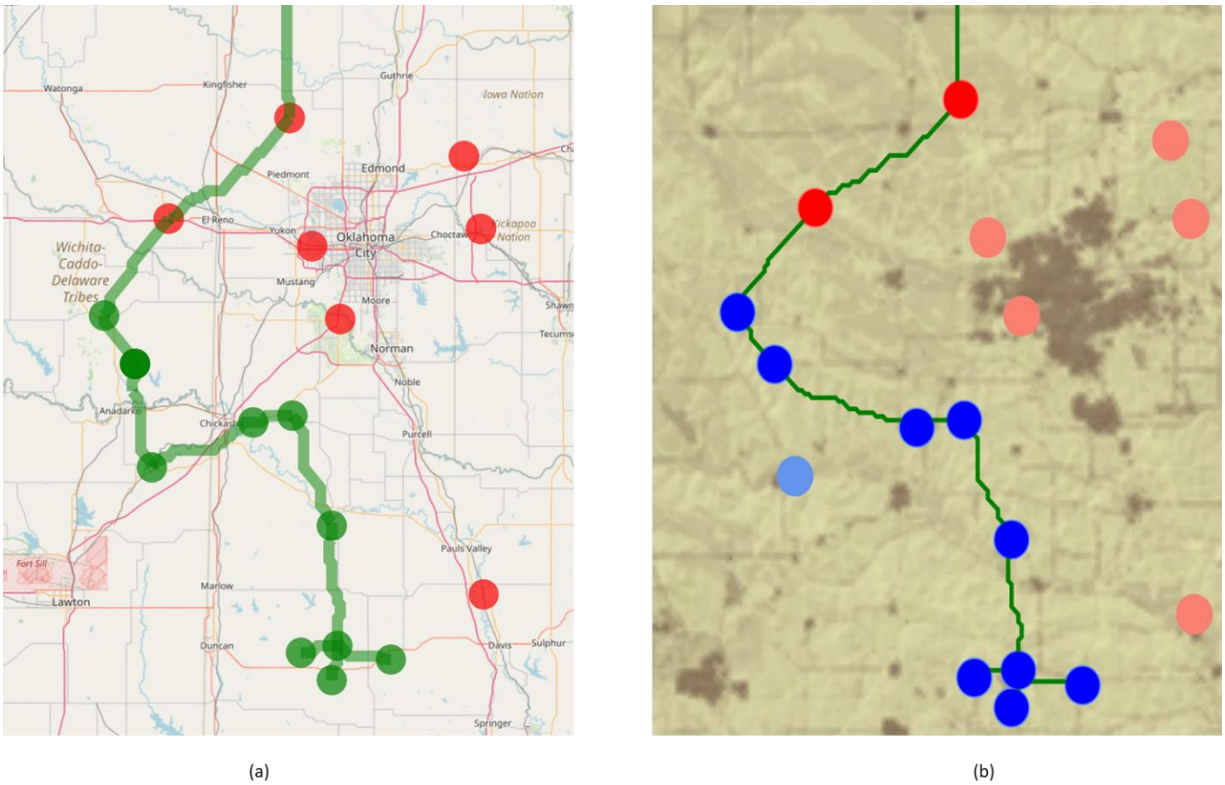



*Figure 33: Zoomed in image of LOWER pipeline path for optimal solutions generated by Sequestrix (a) and SimCCS (b).*

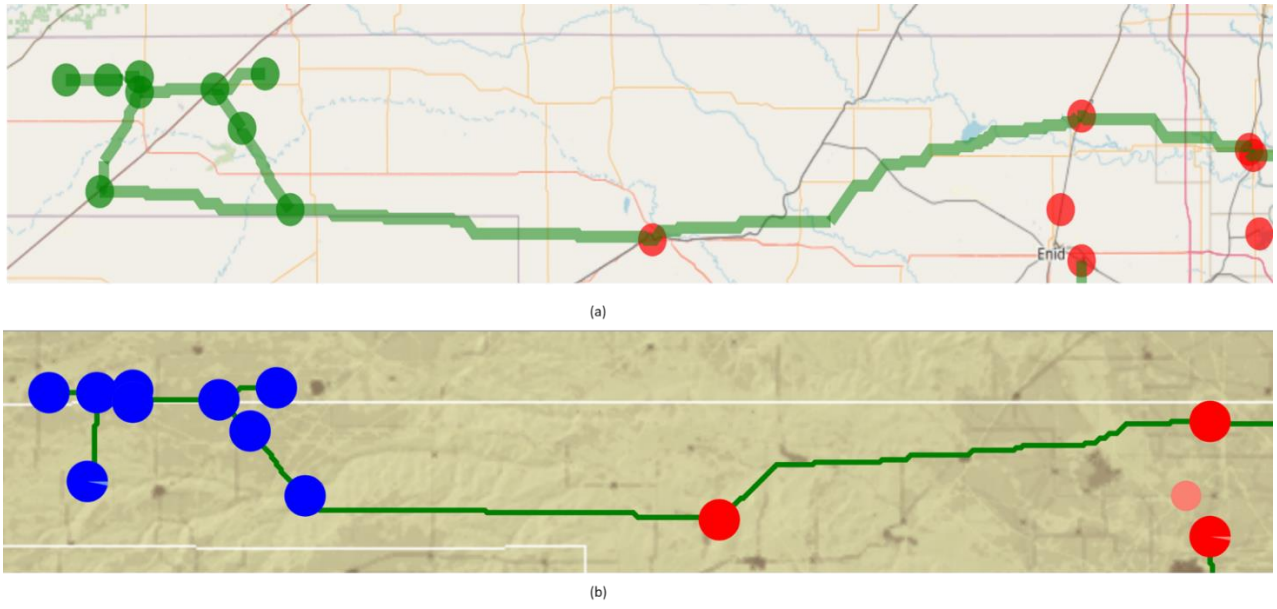



*Figure 34: Zoomed in image of UPPER pipeline path for optimal solutions generated by Sequestrix (a) and SimCCS (b).*

## 4.3 Demo 3 – New Features, Adding Enid-Purdy Pipeline to $CO_2$ Optimization Network

Demo 3 showcases the significant accomplishments of this research project which is incorporating existing pipelines into the cost surface graph and solving optimal $CO_2$ sequestration routes. This demo will concentrate solely on Sequestrix, as SimCCS currently lacks a mechanism for importing existing pipelines into the network.

### 4.3.1 Enid Purdy Pipeline

The US Department of Energy (DOE), in collaboration with the National Energy Technology Laboratory (NETL), published a report (Callahan et al., 2014) detailing the existing $CO_2$ infrastructure in the United States. In the mid-continent region, there are five major pipelines located in Oklahoma and lower Kansas that supply $CO_2$ from various industrial sources to oil and gas operators for use in $CO_2$-EOR. Figure 35 presents a map highlighting these five $CO_2$ pipelines, and Table 9 provides a summary of their operators, lengths, and capacities.

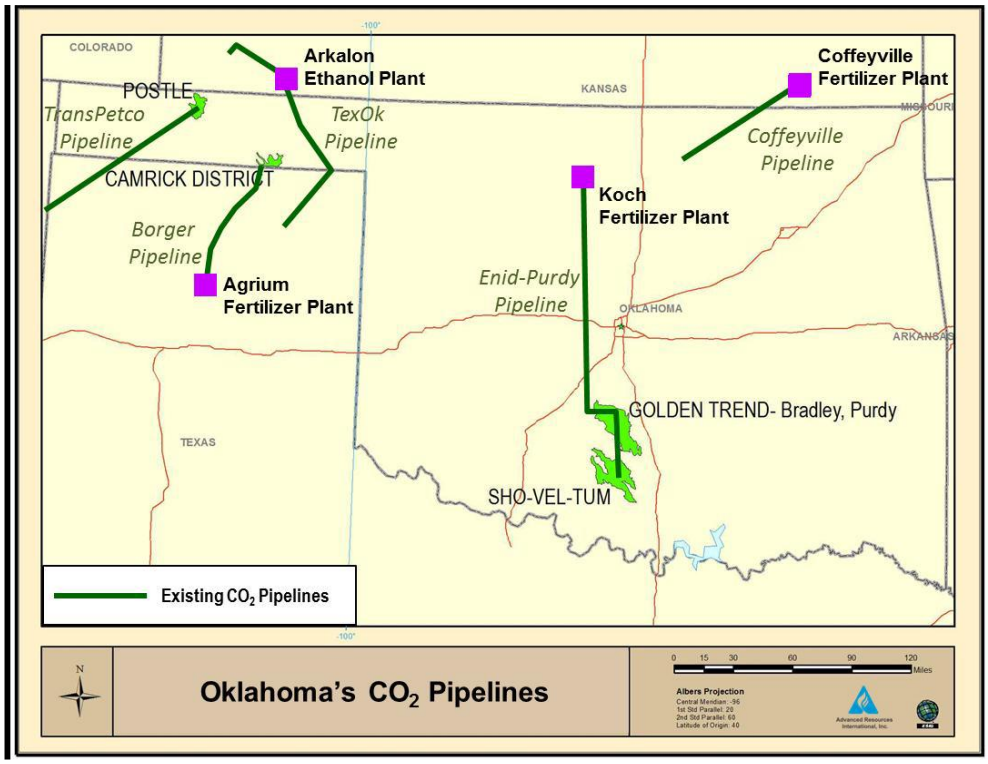



*Figure 35: Mid-Continent $CO_2$ pipeline infrastructure spanning Oklahoma and lower Kansas (Callahan et al., 2014)*

*Table 9: Ownership details and specifications of Mid-Continent transport pipelines(Callahan et al., 2014)*

| Scale | Pipeline | Operator | Location | Length (mi) | Diameter (in) | Estimated Flow Capacity (MMSCF/D) |
|---|---|---|---|---|---|---|
| Small Scale Distribution Systems | Coffeyville-Burbank | Chaparral Energy | KS, OK | 68 | 8 | 80 |
| | Enid-Purdy (Central Oklahoma) | Anadarko | OK | 117 | 8 | 80 |
| | TransPetco | TransPetco | TX, OK | 110 | 8 | 80 |
| | TexOk | Chaparral Energy | OK | 95 | 6 | 70 |
| | Borger | Chaparral Energy | TX, OK | 86 | 4 | 50 |

To highlight Sequestrix's capability to integrate existing pipelines into network optimization, the Enid-Purdy Pipeline in central Oklahoma was selected. The primary reason for this choice is that, as observed in Demo 2, there are numerous $CO_2$ sources and sinks in the surrounding area, and the pipeline is strategically positioned to facilitate thorough utilization of the various scenarios that Sequestrix can handle.

### 4.3.2 $CO_2$ Sources and Sinks Dataset

A subset of the sinks and sources used for demo 2 were selected for demo 3. Additional Sinks representing oil and gas fields the Enid-Purdy pipeline currently injects $CO_2$ into is also included. A detailed list of sources, sinks, their geo-locations, and capacities are given in tables 10 and 11.

*Table 10: Demo 2 $CO_2$ sources information*

| DEMO 3 SOURCES | | | | | |
|---|---|---|---|---|---|
| **ID** | **UNIQUE NAME** | **Capture Capacity (MTCO2/yr)** | **Total Unit Cost ($/tCO2)** | **Lat** | **Lon** |
| 1 | OXBOW CALCINING LLC | 0.32 | 56 | 36.545 | -97.850 |
| 2 | Mustang | 0.53 | 75 | 35.471 | -97.673 |
| 3 | WYNNEWOOD REFINING CO | 0.63 | 75 | 34.629 | -97.169 |
| 4 | Redbud Power Plant | 2.30 | 75 | 35.685 | -97.224 |
| 5 | Horseshoe Lake | 0.60 | 75 | 35.509 | -97.179 |
| 6 | Cana Gas Plant | 0.10 | 14 | 35.535 | -98.099 |
| 7 | OHL NGLP Medford Plant | 0.10 | 14 | 36.776 | -97.756 |
| 8 | TERRA INTERNATIONAL (OKLAHOMA) INC | 0.20 | 17 | 36.437 | -99.471 |

*Table 11: Demo 3 $CO_2$ sinks information*

| DEMO 3 SINKS | | | | | |
|---|---|---|---|---|---|
| **ID** | **UNIQUE NAME** | **Storage Capacity (MTCO2)** | **Total Unit Cost ($/tCO2)** | **Lat** | **Lon** |
| 1 | Field Outlet1 | 4 | -31 | 34.381 | -97.749 |
| 2 | Purdy Field | 3.4 | -31 | 34.758 | -97.605 |
| 3 | i1-i3 | 0.33 | -31 | 35.184 | -98.201 |
| 4 | i10 | 0.12 | -31 | 34.482 | -97.707 |
| 5 | i11 | 0.12 | -31 | 34.506 | -97.598 |
| 6 | Cluster 6 | 3.24 | -31 | 34.473 | -97.441 |
| 7 | Cluster 7 | 2.49 | -31 | 34.421 | -97.614 |
| 8 | Cluster 4 | 1.8 | -31 | 35.300 | -98.295 |

In summary, there are 8 sources with a combined annual capture capacity of 4.77 $MTCO_2$/yr, and 8 sinks with a combined annual storage capacity of 15.5 $MTCO_2$.

### 4.3.3 Base Case – $CO_2$ Network Optimization with No Pipeline

A base case was defined as a standard $CO_2$ network optimization considering the given sources and sinks. This base case will serve as a reference point for comparison when embedding existing pipelines and exploring the various modes available in Sequestrix. The base case was solved for a duration of 10 years, targeting an annual capture of 1.55 $MTCO_2$/yr and a Capital Recovery Factor (CRF) of 10%.

Figure 36 a, b, and c display the Delaunay triangulation results, the alternative new pipeline routes generated, and the selected optimal pipeline built, with all results produced within Sequestrix. Table 12 summarizes the outcome of the base case model.

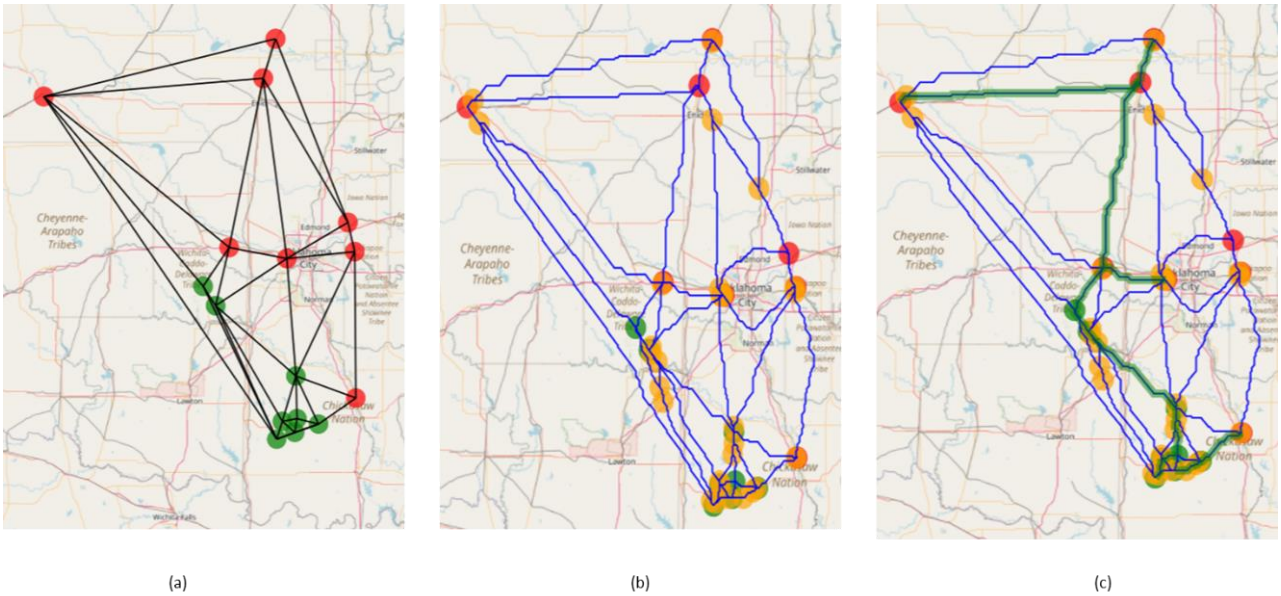



*Figure 36: Demo 3 Base case (no existing pipeline) Sequestrix Solution. (a) Delaunay triangulation results, (b) alternate pipeline routes, (c) Optimal pipeline path selected*

*Table 12: Demo 3 Base Case Sequestrix results*

| Metric | Sequestrix Base Case |
|---|---|
| Unit Capture Cost ($/ton $CO_2$) | 55.77 |
| Unit Transport Cost ($/ton $CO_2$) | 12.39 |
| Unit Storage Cost ($/ton $CO_2$) | -31.00 |
| Unit Total Cost ($/ton $CO_2$) | 37.15 |
| Runtime (seconds) | 108 |
| Total Pipeline Length (km) | 572.56 |

### 4.3.4 Case 1 – Optimization with Enid-Purdy Pipeline 0.5MTCO2/yr Cap No Tie-in No Exclusion

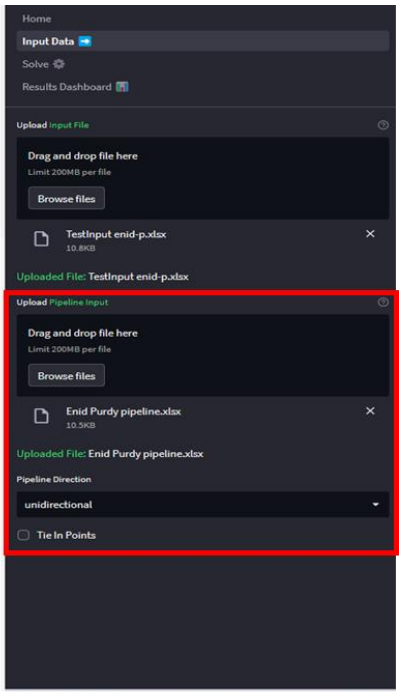
In Case 1, the Enid-Purdy Pipeline is introduced into the cost surface graph for the first time. As described in Section 3, a subset of the pipeline coordinates is extracted from the National Pipeline Mapping System (NPMS) and imported into Sequestrix. This subset is utilized to generate a comprehensive representation of the pipeline through Dijkstra's shortest path augmentation for unconnected edges.

Based on Table 9, the estimated pipeline capacity is 80 MMSCF/D (1.5 $MTCO_2$/yr), and according to partner information, approximately 67% of this capacity is currently in use. Consequently, for the purpose of simulation, the pipeline capacity is set to 0.5 $MTCO_2$/yr to account for the remaining 33%. To utilize this feature in Sequestrix, one simply needs to complete the provided template pipeline file, ensuring that the pipeline geolocations are populated in the direction of the flow through the pipe (upstream to downstream) if it is a unidirectional pipeline. A smaller subset of the Enid-Purdy pipeline used as part of the input file, and the input section detailing where it is uploaded into Sequestrix is displayed in Figure 37 for reference. The resulting source-sink map

with the unrefined pipeline path imported is shown in Figure 38. The process of pipeline location refinement and gap-filling using Dijkstra's algorithm occurs during the solve step, as outlined in Section 3.2.2.

| Lat | Long | Name | Lower Cap | Upper Cap |
|---|---|---|---|---|
| 36.381076 | -97.757562 | enid-purdy-pipeline | 0 | 0.5 |
| 36.381007 | -97.755588 | | | |
| 36.379901 | -97.754301 | | | |
| 36.377276 | -97.754816 | | | |
| 36.355988 | -97.763742 | | | |
| 36.354191 | -97.764686 | | | |
| 35.568233 | -97.788031 | | | |
| 35.524655 | -97.787881 | | | |
| 35.519346 | -97.789061 | | | |
| 35.514875 | -97.786658 | | | |
| 35.434632 | -97.787001 | | | |
| 34.832924 | -97.676892 | | | |
| 34.818833 | -97.668309 | | | |
| 34.812061 | -97.655154 | | | |
| 34.526166 | -97.617341 | | | |
| 34.510891 | -97.623864 | | | |
| 34.493349 | -97.624208 | | | |
| 34.489953 | -97.619744 | | | |
| 34.466746 | -97.619744 | | | |
| 34.447779 | -97.623864 | | | |
| 34.437022 | -97.628327 | | | |
| 34.431074 | -97.629701 | | | |
| 34.431074 | -97.623684 | | | |

(a)




(b)

*Figure 37:(a) Raw Enid-Purdy pipeline input template with latitude, longitude, and capacity specifications. (b) Sequestrix interface for importing existing pipelines*

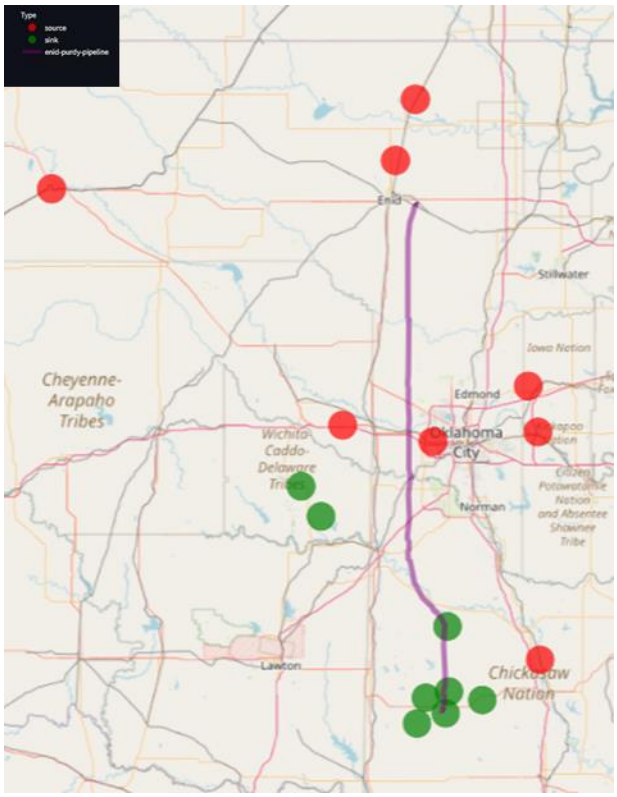



*Figure 38: Sequestrix input page showing map coordinates of the sources and sinks (in red and green respectively) and the raw Enid-Purdy pipeline path (in purple) for Demo 3 case 1*

By default, Sequestrix assumes the pipeline to be unidirectional with no tie-in points or exclusion points, as defined in Section 3. Consequently, during the shortest paths generation, tie-ins into the pipeline are permitted at any point along the pipeline path. This default selection can be modified, and specific tie-in locations and exclusion zones can be specified, as demonstrated in Cases 2 to 6.

The alternative flow network is generated using the same source and sink inputs as well as the Enid-Purdy pipeline, and the new optimization is solved. The results are presented in Figure 39 and Table 13.

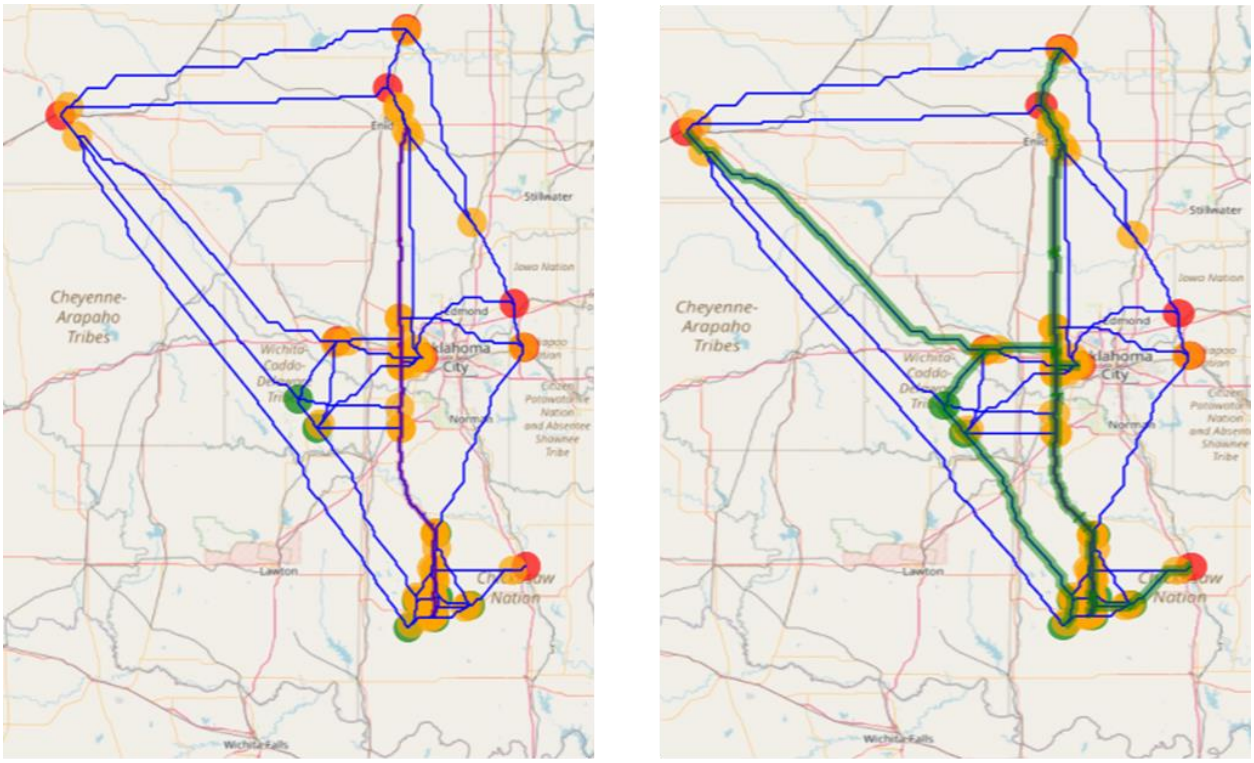



*Figure 39: Sequestrix results for Demo 3, Case 1 - Embedding Enid-Purdy Pipeline with no tie-in locations. (a) Alternate network generated, (b) Optimal solution path passing through existing Pipeline Path*

A qualitative observation reveals that the alternative new pipeline paths generated in Case 2 differ significantly from the results in Case 1, and the selected solution allows some flow to occur along the Enid-Purdy Pipeline, exiting at various points to connect to nearby sinks for sequestration. As shown in Table 13, the unit transportation cost decreases from \$12.39 per ton of $CO_2$ to approximately \$9.91 per ton of $CO_2$ transported, representing an estimated cost reduction of 20%. However, there are limitations to this interpretation. In Sequestrix, the cost surface graph is modified so that an existing pipeline has a total path edge weight of 0. When estimating transportation costs, this edge weight is multiplied by the costs calculated from transportation cost trends, effectively reducing the cost to zero for existing pipelines. In practice, while construction

costs will be zero, operational and maintenance costs will persist, though they are relatively insignificant compared to construction expenses.

*Table 13: Demo 3, Case 1 Sequestrix results*

| Metric | Sequestrix Case 1 |
|---|---|
| **Unit Capture Cost ($/ton $CO_2$)** | 55.77 |
| **Unit Transport Cost ($/ton $CO_2$)** | 9.91 |
| **Unit Storage Cost ($/ton $CO_2$)** | -31.00 |
| **Unit Total Cost ($/ton $CO_2$)** | 34.68 |
| **Runtime (seconds)** | 197 |
| **Enid-Purdy Pipeline Utilized (km)** | 279.68 |
| **Total New Pipeline Length (km)** | 422.03 |

The total length of new pipeline required to achieve the sequestration target also decreases by 27%, from 572 km to 422 km. This reduction is the primary driver of lower transport costs. Based on the results from Case 1, there are seven tie-in points along the pipeline with varying geolocations and connections to different sources and sinks, as illustrated in Figure 40. Case 1's results clearly demonstrate that if an existing pipeline with lower transport costs than a new pipeline is predefined, the optimization algorithm will recommend connecting to this pipeline to transport the maximum amount of $CO_2$ possible.

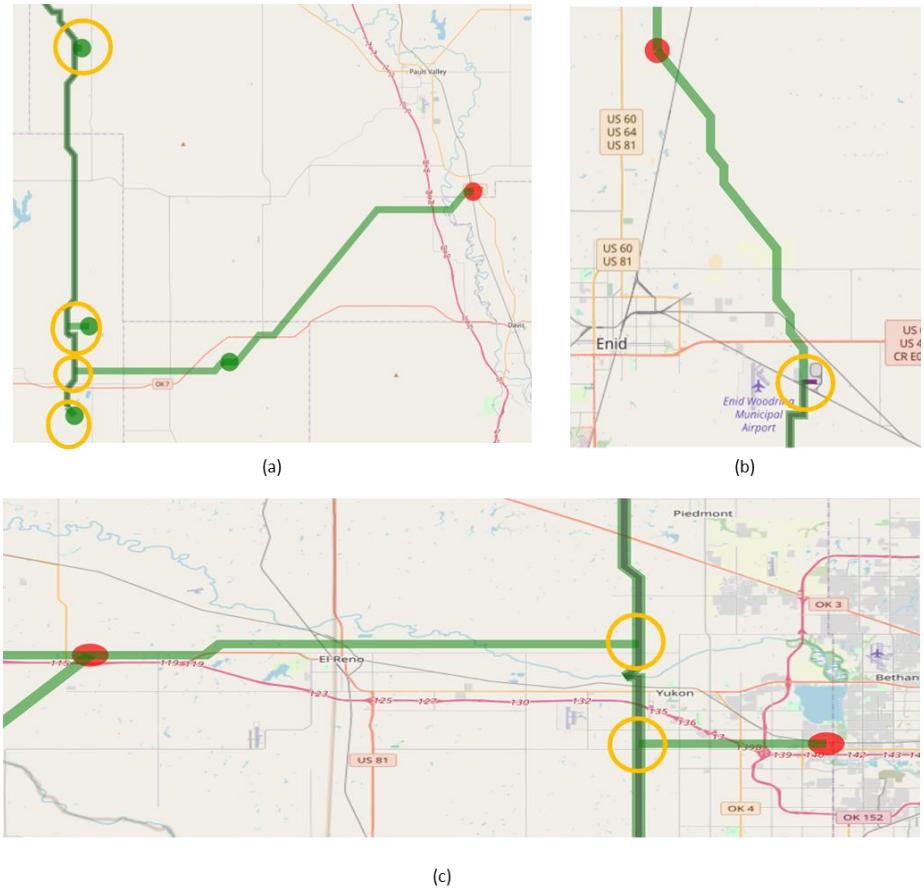



*Figure 40: Zoomed-in results for Demo 3, Case 1, showing all the tie-in points along the Enid-Purdy pipeline suggested by Sequestrix (a) highlights 4 tie-ins with one being an inlet point and the rest being outlet points, (b) Tie-in point towards the begiing of the pipeline facility at Koch Fertiizer plant (c) 2 incoming Tie-in points along Enid-Purdy pipeline path*

### 4.3.5 Case 2 – Enid-Purdy Pipeline 2MTCO$_2$/yr Cap 2 Tie-in points No Exclusion

Case 2 depicts a situation in which a pipeline operator has designated only two tie-in locations along the entire pipeline. This scenario more accurately reflects real-world circumstances, as there are often limitations on the number of tie-in locations an operator is willing to accommodate along a pipeline, which may be due to topographical, environmental, or land rights constraints.

As before, the Enid-Purdy pipeline is utilized in this case; however, the annual transport capacity is now set at 2 $MTCO_2$/yr. The purpose of increasing the capacity beyond the known (or estimated) capacity is to test the optimization limits, promoting the pipeline as the primary transport path. The specified tie-in locations are Lat 35° 57' 1.7" N, Lon -97° 47' 30.58" W, in the northern section of the pipeline, and Lat 35° 0' 42.03" N, -97° 46' 52.54" W in the middle to lower pipeline section. These tie-in points along the pipeline can be entered into the Sequestrix sidebar, as demonstrated in Figures 41 and 42 below.

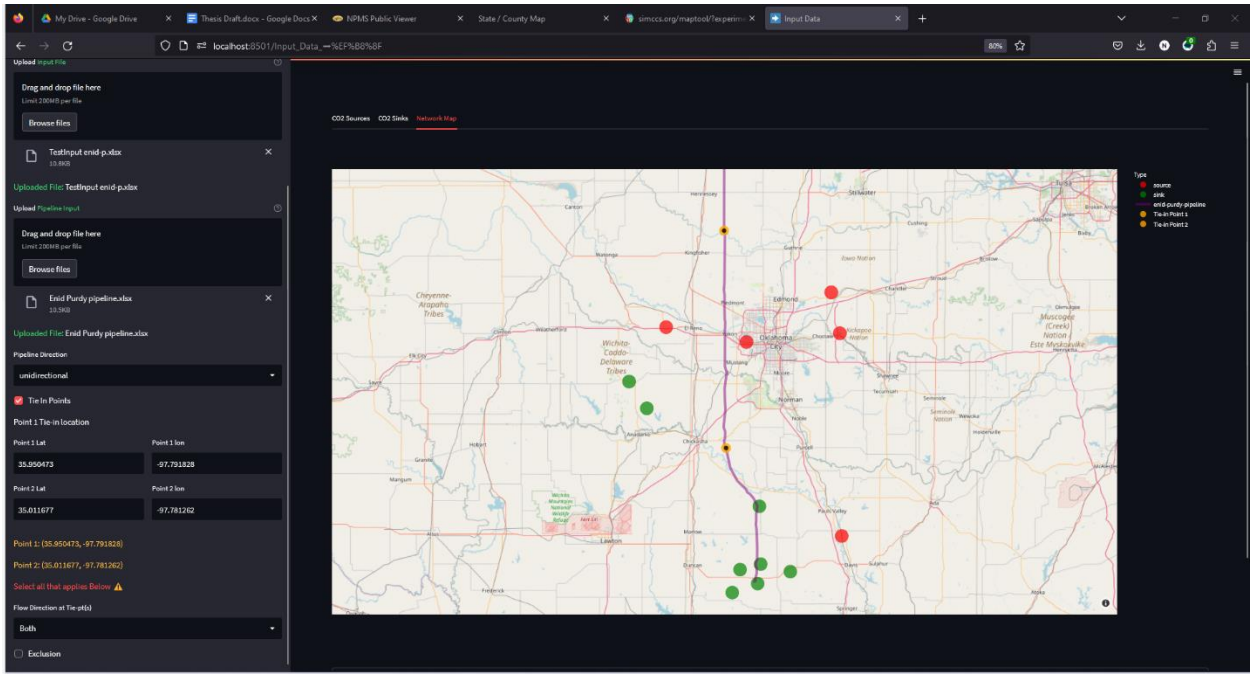



*Figure 41: Sequestrix input page showing tie-in points that were entered on the left sidebar plotted along the Enid-Purdy pipeline.*

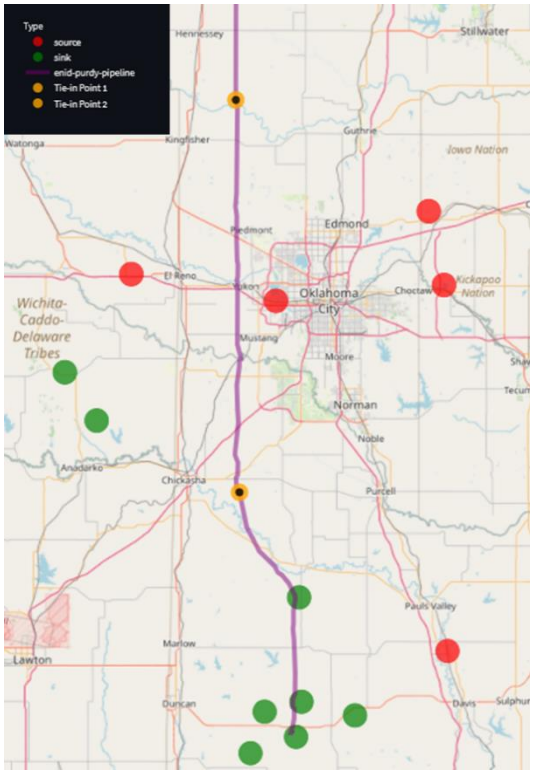



*Figure 42: Zoomed in view of the Enid-Purdy pipeline and 2 tie-in points(colored yellow) with surrounding sources and sinks (colored red and green)*

Upon specifying the tie-in points and rerunning the optimization, a summary of the results is presented in Table 14. In Figure 43, it is evident that the Enid-Purdy pipeline remains unused when specific tie-in locations are established. Furthermore, since the pipeline cannot be accessed from any other location except the designated tie-in points, the proposed paths connecting sources in the north to sinks in the south follow a lengthier route, resulting in a total new pipeline distance that is longer than the base case (575 km vs. 572 km). This extended pipeline consequently impacts the unit transport cost, which is also higher than the base case. In summary, setting specific tie-in locations along the pipeline without allowing for variation proves to be counterproductive in enhancing sequestration economics.

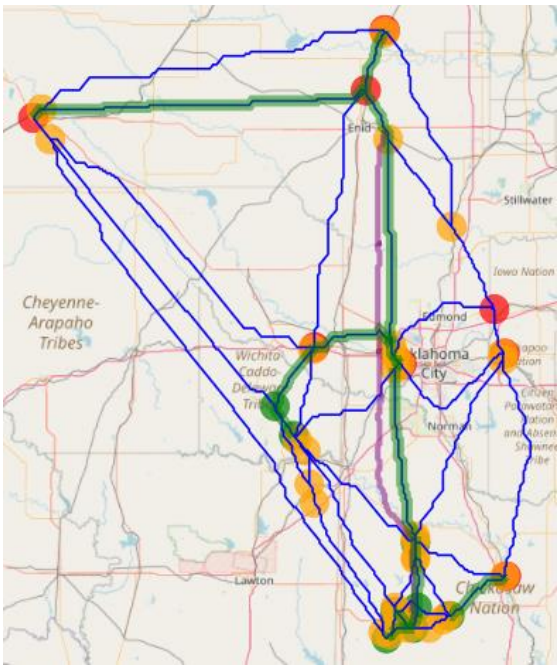



*Figure 43: Resulting Optimal pipeline generated by Sequestrix for Demo 3 Case 2*

*Table 14: Sequestrix Summary of Results for Demo 3 Case 2*

| Metric | Sequestrix Case 2 |
|---|---|
| **Unit Capture Cost ($/ton $CO_2$)** | 55.77 |
| **Unit Transport Cost ($/ton $CO_2$)** | 12.69 |
| **Unit Storage Cost ($/ton $CO_2$)** | -31.00 |
| **Unit Total Cost ($/ton $CO_2$)** | 37.45 |
| **Runtime (seconds)** | 237 |
| **Enid-Purdy Pipeline Utilized (km)** | 0 |
| **Total New Pipeline Length (km)** | 575.32 |

### 4.3.6 Case 3 – Enid-Purdy Pipeline 2MTCO$_2$/yr Cap 2 Tie-in pts Exclusion at Ends

Here the operator specifies 2 geolocations defining a region where any tie-in is allowed. This refers to scenario 2 discussed in section 3.2.3. The tie-in point geolocations chosen for this case are Lat

36° 18' 53.71", Lon -97° 46' 58.06" for point 1 and Lat 36° 39' 27.32”, Lon -97° 46' 58.06" for point 2. The results are summarized in table 15 and figure 44.

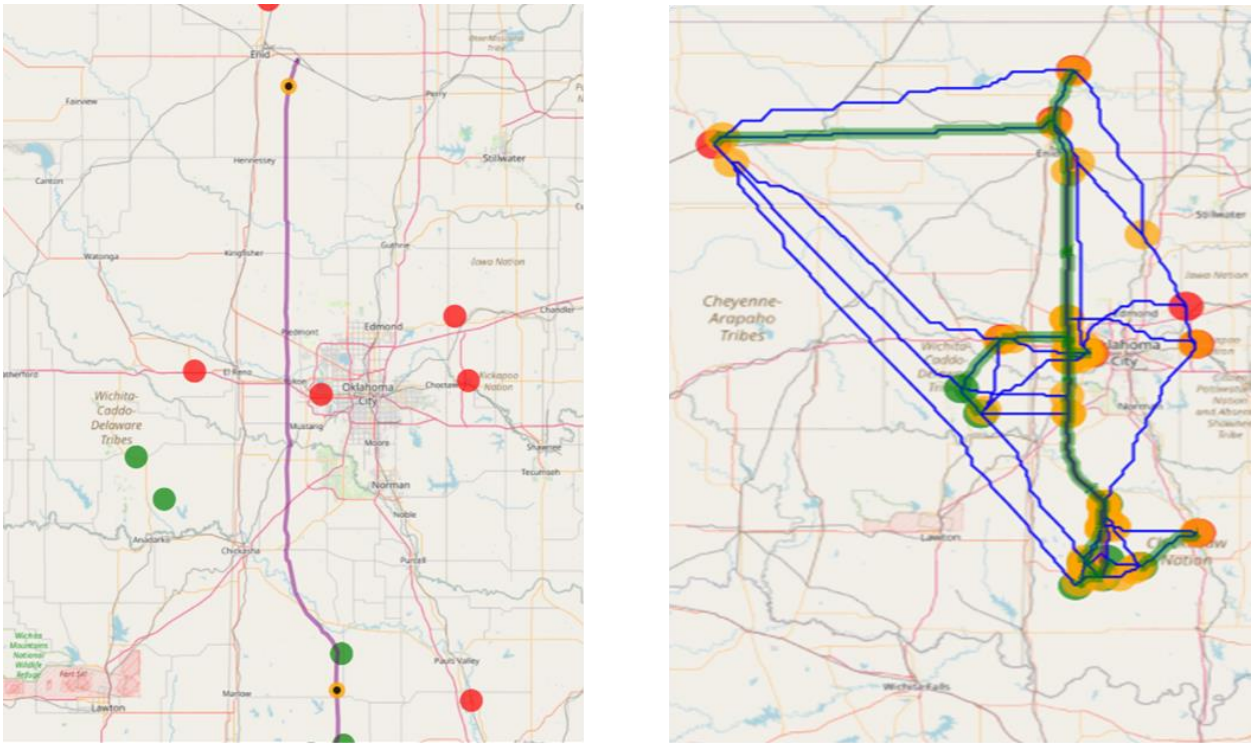



*Figure 44: Sequestrix Input and Solve page map plots for Demo 3 Case 3. (a) shows the Enid-Purdy pipeline with the tie-in points specified. This time an exclusion zone before the tie-in points are activated, (b) shows the optimal pipeline network generated which utilizes the pipeline route.*

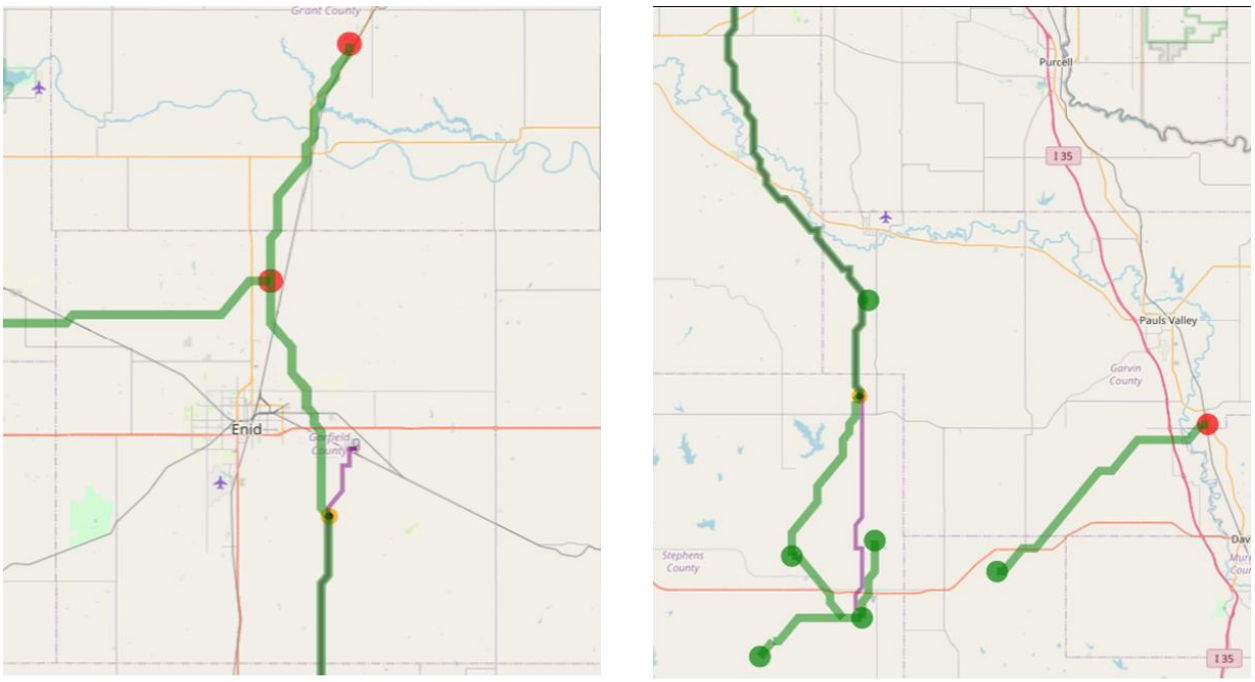



*Figure 45: Zoomed in plot of Demo 3 Case 3 showing that the exclusion zones above and below the 2 tie-in points are honored by Sequestrix*

*Table 15: Sequestrix Summary of Results for Demo 3 Case 3*

| Metric | Sequestrix Case 3 |
|---|---|
| **Unit Capture Cost ($/ton $CO_2$)** | 55.77 |
| **Unit Transport Cost ($/ton $CO_2$)** | 7.80 |
| **Unit Storage Cost ($/ton $CO_2$)** | -31.00 |
| **Unit Total Cost ($/ton $CO_2$)** | 32.57 |
| **Runtime (seconds)** | 204 |
| **Enid-Purdy Pipeline Utilized (km)** | 258.3 |
| **Total New Pipeline Length (km)** | 323.45 |

Figure 45 a and b display a magnified view of the solution map, highlighting that Sequestrix adheres to the specified constraints and that no pipeline tie-in occurs before or after the designated

points 1 and 2. In Case 3, the transport cost associated with utilizing 258 km of the Enid-Purdy pipeline is reduced, which is notably lower than the scenario where the entire pipeline length is available for tie-in. This outcome can be attributed to the fact that during the generation of shortest paths between paired nodes (source-sink, source-source, or sink-sink), the overall transportation cost and potential transport volume are not considered, as they are only optimized after alternative paths have been proposed. Dijkstra's algorithm solely minimizes the total edge weight, which could be seen as a greedy approach to problem-solving. Consequently, exiting the pipeline at a higher (relative to downstream exit of pipeline) location along the path may not yield the lowest edge weights but might facilitate better connections between sources and sinks. This observation becomes evident only after the MIP network optimization problem has been resolved.

A 37% reduction in the unit transportation cost and a 12% decrease in the overall sequestration cost can be observed, which is associated with a shorter length of new pipeline proposed that still effectively delivers the target sequestration volume. This cost reduction is also due to the upgraded pipeline capacity of the base case (from 0.5 to 2 $MTCO_2/yr$) which allows for more flow to be routed through existing pipeline

### 4.3.7 Case 4 – Enid-Purdy Pipeline 2$MTCO_2$/Yr Cap Single Tie-In Point with Exclusion Before

Here the operator has only one single point along the pipeline, beyond which no tie in is allowed. The geolocation of this tie in point is Lat 36° 40' 27.42" N and Lon -97° 47' 2.04" W. Sequestrix comfortably solves this case, and the results are shown in table 16 and figure 46 below:

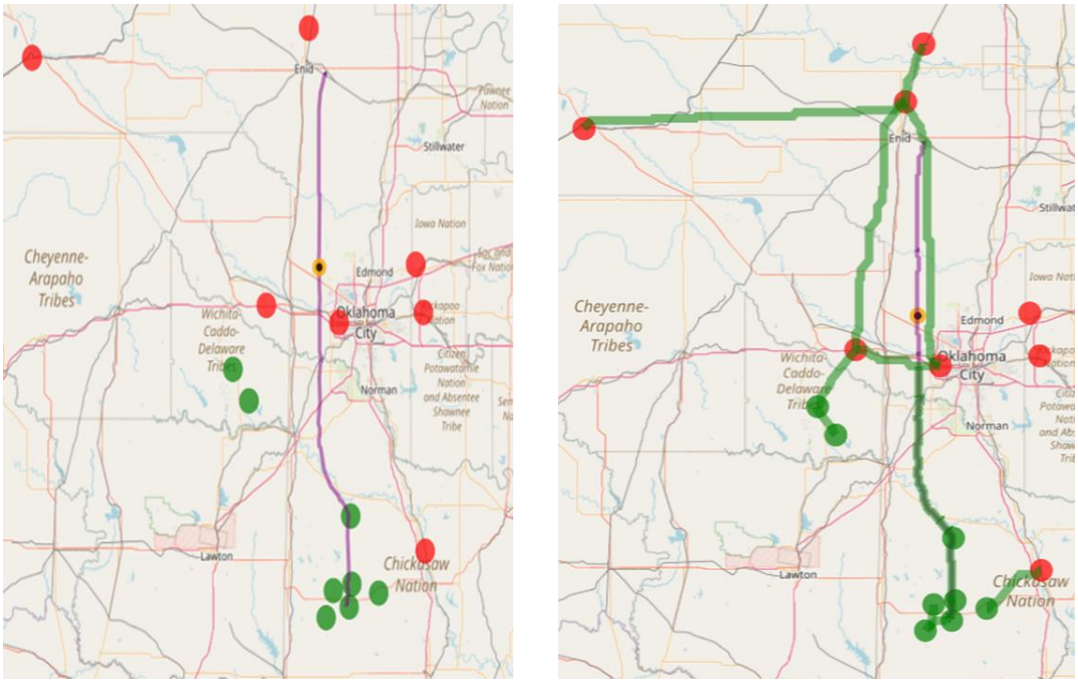



*Figure 46: Zoomed in plot of Demo 3 Case 4 showing that the exclusion zones above and below the single tie-in point is honored by Sequestrix.*

*Table 16: Sequestrix Summary of Results for Demo 3 Case 4*

| Metric | Sequestrix Case 4 |
|---|---|
| **Unit Capture Cost ($/ton $CO_2$)** | 55.77 |
| **Unit Transport Cost ($/ton $CO_2$)** | 9.43 |
| **Unit Storage Cost ($/ton $CO_2$)** | -31.00 |
| **Unit Total Cost ($/ton $CO_2$)** | 34.20 |
| **Runtime (seconds)** | 210.66 |
| **Enid-Purdy Pipeline Utilized (km)** | 138.07 |
| **Total New Pipeline Length (km)** | 567.25 |

A notable increase in the total length of new pipeline employed can be observed, approaching the original length. Nevertheless, the overall cost remains significantly lower than the base case, as a

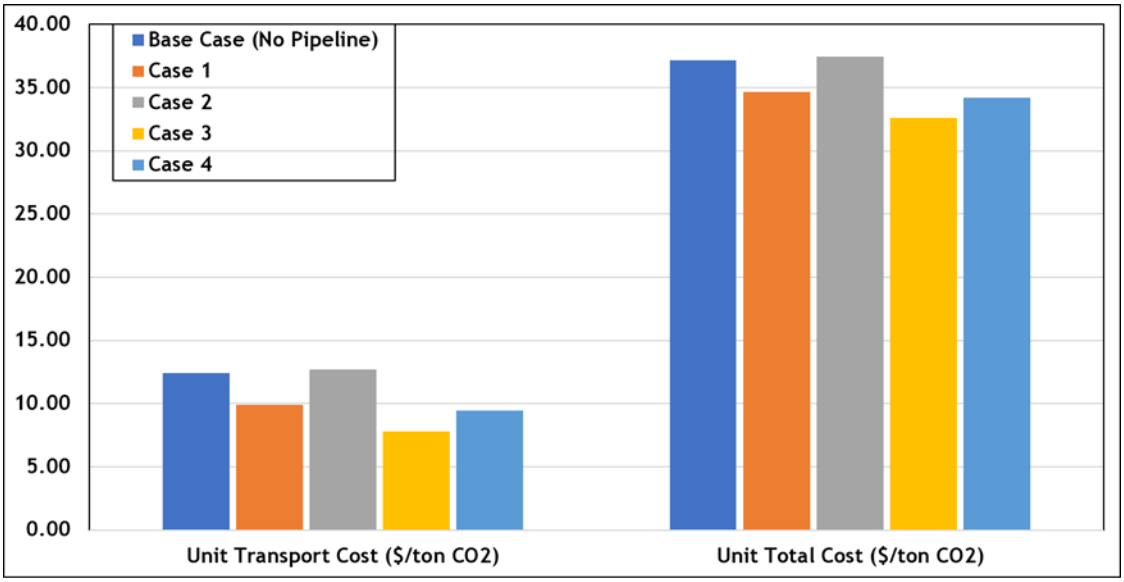
portion of the flow is transported through the 138 km of the Enid-Purdy pipeline. This occurs because a new pipeline is constructed parallel to the existing Enid-Purdy pipeline along the excluded zone length, to convey the necessary volume of $CO_2$ to the storage sites. The choice of a single tie-in point with an exclusion zone preceding the point proves to be ineffective for this demo case.

### 4.3.8 Summary of Embedding Pipelines in $CO_2$ Sequestration Network Optimization

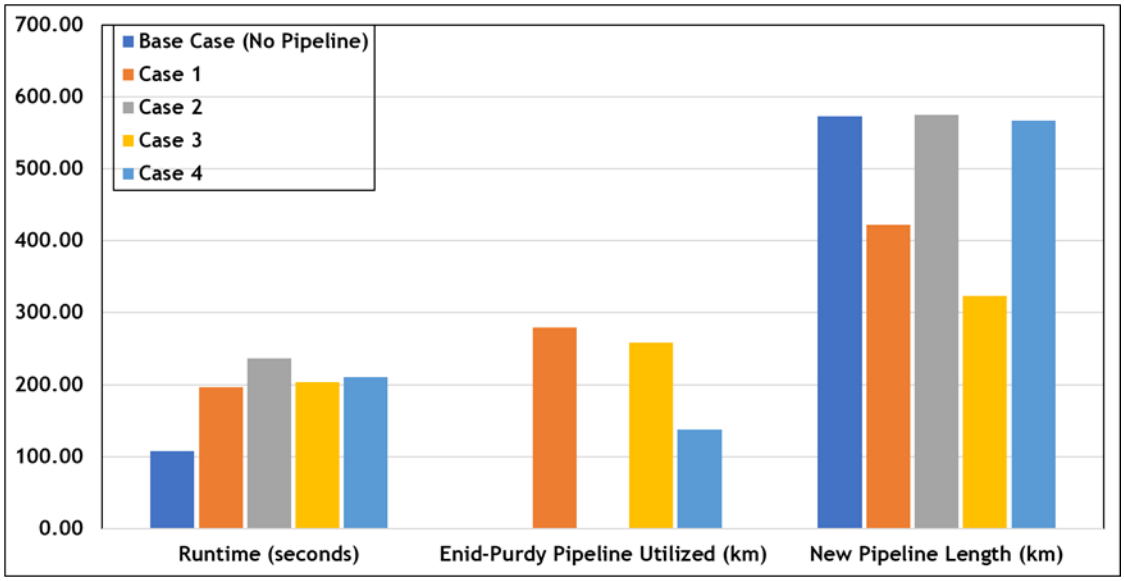
After creating four separate scenarios (1-4), each involving various interactions with the Enid-Purdy pipeline as dictated by the input data and demonstrating Sequestrix's capability to manage multiple tie-in points based on user preferences or limit access to specific areas along the pipeline, a thorough comparison with the base case (excluding the pipeline) can be summarized as follows:

1. Assigning specific tie-in points along an existing pipeline may result in a suboptimal solution compared to a scenario without an integrated existing pipeline, as these tie-in points may not be utilized during the Dijkstra's LCP process.
2. Allowing Sequestrix to identify the optimal tie-in locations without constraints or designating tie-in points with exclusion zones is an effective approach to leverage existing pipeline routes.
3. A general increase in runtime (2x in Demo 3, cases 1-4) is observed when incorporating an existing pipeline, which can be attributed to the preprocessing of sparse pipeline latitude and longitude points and interpolation using Dijkstra's shortest path algorithm.

4. Employing tie-in points with exclusion zones can decrease transportation costs by up to 37%, resulting in an overall cost reduction of 12%.




(a)




(b)

*Figure 47: Overall Comparison plots for Demo 3. (a) shows varying how the transport cost from base case to case 4 affects the total unit cost for project, (b) plots other metrics such as runtime, existing pipeline utilization and new pipeline length proposed.*

# CHAPTER 5: Conclusions

## 5.1 Concluding Remarks

The primary aim of this research was to develop a method for incorporating existing pipelines into $CO_2$ sequestration network optimization, for which six algorithms were devised. The initial algorithm presents a technique for adjusting the edge weights of the cost surface graph, setting them to zero for grid cells representing the geolocation of a pipeline route. The subsequent four algorithms enable users to designate tie-in points and exclusion zones along any given pipeline, managing the ingress and egress of new pipeline streams. The final algorithm restricts diagonal crossover along existing pipeline routes.

These six algorithms, combined with supplementary path post-processing algorithms delineated in the APPENDIX, and established algorithms from SimCCS, the preeminent platform for techno-economic $CO_2$ sequestration network optimization, facilitated the creation of a novel software package called Sequestrix. Developed using the Python programming language and various open-source front-end applications, Sequestrix offers users the flexibility to assess multiple sequestration scenarios and obtain verifiably accurate estimates of capture, transport, and storage costs for CCS projects.

To ensure the software's reliability, Sequestrix was subjected to benchmark testing against SimCCS on a local computer. The results demonstrated a considerable improvement in performance, with a 68% reduction in runtime. These enhancements can be partially attributed to solver selection (Gurobi over CPLEX) but primarily to the manner in which the input cost surface

graph is processed. Importantly, these improvements did not compromise the quality of Sequestrix's output, as discussed in the preceding section.

Sequestrix Demo 3 illustrated various scenarios for assigning or calculating tie-in points along the Enid-Purdy pipeline. Compared to the base case without the pipeline, utilizing the pipeline can result in up to a 12% decrease in the unit total cost of sequestration. However, specifying single tie-in points without exclusion zones may adversely affect sequestration economics, potentially leading to an increase in unit total sequestration costs compared to the base case, however this can only be verified for the cases tested.

Currently, Sequestrix is hosted on GitHub and can be accessed at https://github.com/davidpcg01/$CO_2$-TRANSPORT-NETWORK-OPTIMIZATION-PROJECT. The choice to develop Sequestrix in Python was deliberate, enabling straightforward deployment on multiple local computers, with all required components installable via the pip installer for Python. In the future, the software may be made publicly available on a dedicated website.

## 5.2 Future Work

To further extend this work, researchers may consider modifying the cost along existing pipeline path to a fixed user input number to account for operational cost or utilizing a new set of linearized pipeline operational costs which will only be applied to transmission nodes along an existing pipeline. In doing this, care must be taken to ensure it is applied after initially defining the zero-cost path and finding all LCP that will traverse the tie-in points and exclusion zones.

Also, the diagonal pipeline exclusion algorithm may be applied on new paths whilst generating LCP, however an optimized heuristic is required determine the sequence in which these LCPs are

generated, one suggested way is to assign importance weights to nodepair connections, the paths can then be generated in ranked order of importance.

To account for geologic uncertainty related to the volume of $CO_2$ that can be stored, when running optimization, one may use probabilistic storage capacities with assigned likelihood of actually meeting that target. Designing alternate pipeline networks can now be done in a way such that the routes generated will honor the target sequestration volume by function of storage volume and likelihood of success.

Finally, this theoretical background behind the development of optimized transport networks and embedding existing pipelines can also be transferred to solving similar routing challenges for hydrogen storage, which is crucial to energy transition. In this case, the sources and the sinks geo information must be modified to represent physical hydrogen generation plants, and underground storage facilities.

## References


Abramson, E., McFarlane, D., & Brown, J. (2020). *WHITEPAPER ON REGIONAL INFRASTRUCTURE FOR MIDCENTURY DECARBONIZATION*.

Achanta, D. S., Balch, R. S., & Grigg, R. B. (2012, February 7). *Simulation of Leakage Scenarios for CO2 Storage at Gordon Creek, Utah*. Carbon Management Technology Conference. https://doi.org/10.7122/151483-MS

Ajoma, E., Sungkachart, T., Saira, -, Yin, H., & Le-Hussain, F. (2021). A Laboratory Study of Coinjection of Water and CO2 to Improve Oil Recovery and CO2 Storage: Effect of Fraction of CO2 Injected. *SPE Journal*, *26*(04), 2139–2147. https://doi.org/10.2118/204464-PA

Beckwith, R. (2011). Carbon Capture and Storage: A Mixed Review. *Journal of Petroleum Technology*, *63*(05), 42–45. https://doi.org/10.2118/0511-0042-JPT

Bielicki, J. M., Calas, G., Middleton, R. S., & Ha-Duong, M. (2014). National corridors for climate change mitigation: Managing industrial CO2 emissions in France. *Greenhouse Gases: Science and Technology*, *4*(3), 262–277. https://doi.org/10.1002/ghg.1395

Bielicki, J. M., Langenfeld, J. K., Tao, Z., Middleton, R. S., Menefee, A. H., & Clarens, A. F. (2018). The geospatial and economic viability of CO2 storage in hydrocarbon depleted fractured shale formations. *International Journal of Greenhouse Gas Control*, *75*(C), Article LA-UR-17-24674. https://doi.org/10.1016/j.ijggc.2018.05.015

Burghardt, J., & Appriou, D. (2021, June 18). *State of Stress Uncertainty Quantification and Geomechanical Risk Analysis for Subsurface Engineering*. 55th U.S. Rock Mechanics/Geomechanics Symposium. https://onepetro.org/ARMAUSRMS/proceedings-abstract/ARMA21/All-ARMA21/468335

Callahan, K., Goudarzi, L., Wallace, M., & Wallace, R. (2014). *A Review of the CO2 Pipeline Infrastructure in the U.S.* (DOE/NETL-2014/1681). National Energy Technology Lab. (NETL), Albany, OR (United States). https://doi.org/10.2172/1487233

Christensen, J. R., Stenby, E. H., & Skauge, A. (2001). Review of WAG Field Experience. *SPE Reservoir Evaluation & Engineering*, *4*(02), 97–106. https://doi.org/10.2118/71203-PA

DaneshFar, J., Nnamdi, D., Moghanloo, R. G., & Ochie, K. (2021). Economic Evaluation of CO2 Capture, Transportation, and Storage Potentials in Oklahoma. *Day 1 Tue, September 21, 2021*, D011S003R002. https://doi.org/10.2118/206106-MS

Delaunay, B. (1934). Sur la sphère vide. A la mémoire de Georges Voronoï. *Bulletin de l'Académie des Sciences de l'URSS. Classe des sciences mathématiques et na*, *6*, 793–800.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271. https://doi.org/10.1007/BF01386390

Driezen, K., Adriaensen, F., Rondinini, C., Doncaster, C. P., & Matthysen, E. (2007). Evaluating least-cost model predictions with empirical dispersal data: A case-study using radiotracking data of hedgehogs (Erinaceus europaeus). *Ecological Modelling*, *209*(2), 314–322.

Ellett, K. M., Middleton, R. S., Stauffer, P. H., & Rupp, J. A. (2017). Facilitating CCS Business Planning by Extending the Functionality of the SimCCS Integrated System Model. *Energy Procedia*, *114*, 6526–6535. https://doi.org/10.1016/j.egypro.2017.03.1788

Fakher, S., & Imqam, A. (2019). A review of carbon dioxide adsorption to unconventional shale rocks methodology, measurement, and calculation. *SN Applied Sciences*, *2*(1), 5. https://doi.org/10.1007/s42452-019-1810-8

Gale, J., Christensen, N. P., Cutler, A., & Torp, T. A. (2001). Demonstrating the Potential for Geological Storage of CO2: The Sleipner and GESTCO Projects. *Environmental Geosciences*, *8*(3), 160–165. https://doi.org/10.1046/j.1526-0984.2001.008003160.x

Gorucu, F. B., Jikich, S. A., Bromhal, G. S., Sams, W. N., Ertekin, T., & Smith, D. H. (2005). *Matrix shrinkage and swelling effects on economics of enhanced coalbed methane production and CO 2 sequestration in coal: Society of Petroleum Engineers Eastern Regional Meeting 2005*. http://www.scopus.com/inward/record.url?scp=84858562446&partnerID=8YFLogxK

Gu, Y., & Yang, D. (2004). Interfacial Tensions and Visual Interactions of Crude Oil-Brine-CO Systems Under Reservoir Conditions. *Canadian International Petroleum Conference*. Canadian International Petroleum Conference, Calgary, Alberta. https://doi.org/10.2118/2004-083

Hepburn, C., Adlen, E., Beddington, J., Carter, E. A., Fuss, S., Mac Dowell, N., Minx, J. C., Smith, P., & Williams, C. K. (2019). The technological and economic prospects for CO2

utilization and removal. *Nature*, *575*(7781), 87–97. https://doi.org/10.1038/s41586-019-1681-6

Hoover, B., Middleton, R. S., & Yaw, S. (2019). *CostMAP: An open-source software package for developing cost surfaces*.

Hopkins, L. D. (1973). Design method evaluation—An experiment with corridor selection. *Socio-Economic Planning Sciences*, *7*(5), 423–436.

Huber, D. L., & Church, R. L. (1985). TRANSMISSION CORRIDOR LOCATION MODELING. *Journal of Transportation Engineering*, *111*(2), Article Reprint. https://trid.trb.org/view/270278

IEA. (2022). *World Energy Outlook*.

IPCC. (2005). *IPCC Special Report on Carbon Dioxide Capture and Storage. Prepared by Working Group III of the Intergovernmental Panel on Climate Change [Metz, B., O. Davidson, H. C. de Coninck, M. Loos, and L. A. Meyer (eds.)]*. (p. 442 pp). Cambridge University Press.

IPCC. (2022). *Climate Change 2022: Impacts, Adaptation and Vulnerability. Contribution of Working Group II to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change [H.-O. Pörtner, D.C. Roberts, M. Tignor, E.S. Poloczanska, K. Mintenbeck, A. Alegr*. Cambridge University Press. doi:10.1017/9781009325844

Jones, E. C., Yaw, S., Bennett, J. A., Ogland-Hand, J. D., Strahan, C., & Middleton, R. S. (2022). Designing multi-phased CO 2 capture and storage infrastructure deployments. *Renewable and Sustainable Energy Transition*, *2*, 100023. https://doi.org/10.1016/j.rset.2022.100023

Keil, J. M., & Gutwin, C. A. (1992). Classes of graphs which approximate the complete euclidean graph. *Discrete & Computational Geometry*, *7*(1), 13–28. https://doi.org/10.1007/BF02187821

Lobo, L. J. (2017). *An Improved Mathematical Formulation For the Carbon Capture and Storage (CCS) Problem*. University of Arizona.

Lugschitz, H. (2017). Overhead Lines and Underground Cables. In K. O. Papailiou (Ed.), *Overhead Lines* (pp. 1299–1318). Springer International Publishing. https://doi.org/10.1007/978-3-319-31747-2_19

McPherson, B. J. O. L., & Cole, B. S. (2000). Multiphase CO2 flow, transport and sequestration in the Powder River Basin, Wyoming, USA. *Journal of Geochemical Exploration*, *69–70*, 65–69. https://doi.org/10.1016/S0375-6742(00)00046-7

Middleton, R. S. (2013). A new optimization approach to energy network modeling: Anthropogenic CO2 capture coupled with enhanced oil recovery. *International Journal of Energy Research*, *37*(14), 1794–1810. https://doi.org/10.1002/er.2993

Middleton, R. S., & Bielicki, J. M. (2009). A scalable infrastructure model for carbon capture and storage: SimCCS. *Energy Policy*, *37*(3), 1052–1060. https://doi.org/10.1016/j.enpol.2008.09.049

Middleton, R. S., Bielicki, J. M., Keating, G. N., & Pawar, R. J. (2011). Jumpstarting CCS using refinery CO2 for enhanced oil recovery. *Energy Procedia*, *4*, 2185–2191. https://doi.org/10.1016/j.egypro.2011.02.105

Middleton, R. S., & Brandt, A. R. (2013). Using Infrastructure Optimization to Reduce Greenhouse Gas Emissions from Oil Sands Extraction and Processing. *Environmental Science & Technology*, *47*(3), 1735–1744. https://doi.org/10.1021/es3035895

Middleton, R. S., Clarens, A. F., Liu, X., Bielicki, J. M., & Levine, J. S. (2014). CO2 Deserts: Implications of Existing CO2 Supply Limitations for Carbon Management. *Environmental Science & Technology*, *48*(19), 11713–11720. https://doi.org/10.1021/es5022685

Middleton, R. S., Keating, G. N., Stauffer, P. H., Jordan, A. B., Viswanathan, H. S., Kang, Q. J., Carey, J. W., Mulkey, M. L., Sullivan, E. J., Chu, S. P., Esposito, R., & Meckel, T. A. (2012). The cross-scale science of CO2 capture and storage: From pore scale to regional scale. *Energy & Environmental Science*, *5*(6), 7328–7345. https://doi.org/10.1039/C2EE03227A

Middleton, R. S., Kuby, M. J., & Bielicki, J. M. (2012). Generating candidate networks for optimization: The CO2 capture and storage optimization problem. *Computers, Environment and Urban Systems*, *36*(1), 18–29. https://doi.org/10.1016/j.compenvurbsys.2011.08.002

Middleton, R. S., Yaw, S. P., Hoover, B. A., & Ellett, K. M. (2020). SimCCS: An open-source tool for optimizing CO2 capture, transport, and storage infrastructure. *Environmental Modelling & Software*, *124*, 104560. https://doi.org/10.1016/j.envsoft.2019.104560

Mohamed, I. M., & Nasr-El-Din, H. A. (2012, February 15). *Formation Damage Due to CO2 Sequestration in Deep Saline Carbonate Aquifers*. SPE International Symposium and Exhibition on Formation Damage Control. https://doi.org/10.2118/151142-MS

Morbee, J., Serpa, J., & Tzimas, E. (2011). Optimal planning of CO2 transmission infrastructure: The JRC InfraCCS tool. *Energy Procedia*, *4*, 2772–2777. https://doi.org/10.1016/j.egypro.2011.02.180

Morgan, D., Guinan, A., & Sheriff, A. (2022). *FECM/NETL CO2 Transport Cost Model (2022): Description and User's Manual* (DOE/NETL-2022/3218). National Energy Technology Laboratory (NETL), Pittsburgh, PA, Morgantown, WV, and Albany, OR (United States). https://doi.org/10.2172/1856355

NOAA. (2022). *Climate Change: Atmospheric Carbon Dioxide*. http://www.climate.gov/news-features/understanding-climate/climate-change-atmospheric-carbon-dioxide

Núñez-López, V., & Moskal, E. (2019). Potential of CO2-EOR for Near-Term Decarbonization. *Frontiers in Climate*, *1*. https://www.frontiersin.org/articles/10.3389/fclim.2019.00005

Ochie, K., Burghardt, J., Rouzbeh, M., & Daneshfar, J. (2022, September 26). *A Probability Evaluation of Seismicity Risks Associated with CO2 Injection into Arbuckle Formation*. SPE Annual Technical Conference and Exhibition. https://doi.org/10.2118/210345-MS

Rayfield, B., Fortin, M. J., & Fall, A. (2010). The sensitivity of least-cost habitat graphs to relative cost surface values. *Landscape Ecology*, *25*(4), 519–532.

Rogers, J., & Grigg, R. (2001). A Literature Analysis of the WAG Injectivity Abnormalities in the CO2 Process. *SPE Reservoir Evaluation & Engineering - SPE RESERV EVAL ENG*, *4*, 375–386. https://doi.org/10.2118/73830-PA

SimCCS. (2021). *SimCCS*. https://github.com/simccs/SimCCS

Stauffer, P., Middleton, R., Bing, B., Ellett, K., Rupp, J., & Xiaochun, L. (2014). System integration linking CO2 Sources, Sinks, and Infrastructure for the Ordos Basin, China. *Energy Procedia*, *63*, 2702–2709. https://doi.org/10.1016/j.egypro.2014.11.292

Stucky, J. L. D. (1998). On applying viewshed analysis for determining least-cost paths on Digital Elevation Models. *International Journal of Geographical Information Science*, *12*(8), 891–905. https://doi.org/10.1080/136588198241554

Talsma, C., Middleton, E., & Middleton, R. (2022). Costmappro: Addressing the Massive-Scale Co2 Pipeline Challenge. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.4273192

Tarrahi, M., & Afra, S. (2015). Optimization of Geological Carbon Sequestration in Heterogeneous Saline Aquifers through Managed Injection for Uniform CO2 Distribution. *All Days*, CMTC-440233-MS. https://doi.org/10.7122/440233-MS

Whitman, C., Yaw, S., Hoover, B., & Middleton, R. (2022). Scalable algorithms for designing CO2 capture and storage infrastructure. *Optimization and Engineering*, *23*(2), 1057–1083. https://doi.org/10.1007/s11081-021-09621-3

Yaw, S., Middleton, R. S., & Hoover, B. (2019). Graph Simplification for Infrastructure Network Design. In Y. Li, M. Cardei, & Y. Huang (Eds.), *Combinatorial Optimization and Applications* (Vol. 11949, pp. 576–589). Springer International Publishing. https://doi.org/10.1007/978-3-030-36412-0_47

Yu, C., Lee, J., & Munro-Stasiuk, M. J. (2003). Research Article: Extensions to least-cost path algorithms for roadway planning. *International Journal of Geographical Information Science*, *17*(4), 361–376. https://doi.org/10.1080/1365881031000072645

## NOMENCLATURE

| | |
|---|---|
| CCS | Carbon Capture and Storage |
| $CO_2$ | Carbon Dioxide |
| CUSP | Carbon, Utilization, Storage Partnership |
| EOR | Enhanced Oil Recovery |
| GIS | Geographical Information System |
| IEA | International Energy Agency |
| IPCC | Intergovernmental Panel on Climate Change |
| LCP | Least Cost Path |
| MT | Megatons (Mega metric tons) |
| mT | Metric Tons |
| MIP | Mixed Integer Programming |
| MILP | Mixed Integer Linear Programming |
| MIQP | Mixed Integer Quadratic Programming |
| SimCCS | Scalable Infrastructure Model for CCS |

# APPENDIX

## Sequestrix Source Code

Key code components that power the Sequestrix application is given in this appendix, all the other code and input files are publicly available on GitHub via the following link: https://github.com/davidpcg01/$CO_2$-TRANSPORT-NETWORK-OPTIMIZATION-PROJECT

geotranformation.py

```
import pandas as pd
import numpy as np
from csv import reader
import time
from geopy.point import Point
from geopy.distance import distance
from bisect import bisect_left, bisect_right
from pathlib import Path

ROOT_PATH = Path(__file__).parent.parent.resolve()
# FILE_PATH = ROOT_PATH.joinpath("Construction Costs.csv")
FILE_PATH = ROOT_PATH.joinpath("construction-costs-subset.csv")



class geoTransformation:
    def __init__(self) -> None:
        self.costFilePath = FILE_PATH
        self.gridcost = {}
        self.gridCostList = []
        self.gridTranslated = False
        self.north = 40.422261
        self.south = 33.615165
        self.east = -92.284113
        self.west = -103.665777

    def _loadgeogrid(self) -> None:
        with open(self.costFilePath, 'r') as read_obj:
            csv_reader = reader(read_obj)
            i = 0
            while i < 2:
                next(csv_reader)
                i += 1
            self.gridWidth = int(next(csv_reader)[1])
```

```
            self.gridHeight = int(next(csv_reader)[1])
            self.lowerLeftX = float(next(csv_reader)[1])
            self.lowerLeftY = float(next(csv_reader)[1])
            self.cellSize = float(next(csv_reader)[1])
            self.noDataValue = next(csv_reader)[1]

        self.gridVertices = [i for i in range(1,
((self.gridWidth*self.gridHeight) + 1))]



    def _loadcost(self):

        with open(self.costFilePath, 'r') as read_obj:
            csv_reader = reader(read_obj)
            i = 0
            while i < 8:
                next(csv_reader)
                i += 1
            edgeConn = next(csv_reader)
            while edgeConn != ['']:
                edgeCost = next(csv_reader)
                startnode = int(edgeConn[0])
                for i in range(len(edgeCost)):
                    key = (startnode, int(edgeConn[i+1]))
                    if self._checkBound(key):
                        self.gridcost[key] =  float(edgeCost[i])
                edgeConn = read_obj.readline().split(",")
                edgeConn[-1] = edgeConn[-1].split("\n")[0]



    def create_grid(self):
        nrows = self.gridHeight
        ncols = self.gridWidth
        grid = []
        counter = 1
        for row in range(nrows):
            row_list = []
            for col in range(ncols):
                row_list.append(counter)
                counter += 1
            grid.append(row_list)
        return grid

    def translate_grid(self):
        nrows = self.gridHeight
        ncols = self.gridWidth
        grid = []
        for row in range(nrows):
            row_list = []
            for col in range(ncols):
```

```
                cell_number = (nrows - row) * ncols - col
                row_list.append(cell_number)
            row_list.reverse()
            grid.append(row_list)
        return grid


    def _generateGridCostList(self):
        for key in self.gridcost.keys():
            self.gridCostList.append([key[0], key[1], {'weight':
self.gridcost[key]}])

    def _getNeighbors(self, cell):
        neighbors = [cell+1, cell-1, cell + self.gridWidth, cell -
self.gridWidth, cell + self.gridWidth + 1,
                     cell + self.gridWidth - 1, cell - self.gridWidth + 1, cell -
self.gridWidth - 1]
        for i in range(len(neighbors)):
            if (neighbors[i] < 0) or (neighbors[i] > self.gridHeight *
self.gridWidth):
                neighbors[i] = 0
        return neighbors

    def _initializeCostgrid(self):
        for i in self.gridVertices:
            neighbors = self._getNeighbors(i)
            for neighbor in neighbors:
                if neighbor != 0:
                    self.gridcost[(i, neighbor)] = 1e6


    def _vicenty(self, distance_km, point_a):
        lat_a = point_a[0]
        lon_a = point_a[1]

        # calculate the distance between two points separated by 1 degree of
longitude at point_a's latitude
        lon_degrees_offset = distance(point_a, Point(lat_a, lon_a + 1)).km

        # calculate the distance in degrees to travel to move distance_km east
        lon_degrees_to_travel = distance_km / lon_degrees_offset

        # calculate the longitude of point_b
        lon_b = lon_a + lon_degrees_to_travel

        # return a Point object for point_b
        return Point(latitude=lat_a, longitude=lon_b)

    def _latlonToCell(self, lat, lon):
        y = self.gridHeight - (int((lat - self.lowerLeftY) / self.cellSize) + 1)
+ 1
        x = int((lon - self.lowerLeftX) / self.cellSize) + 1
```

```
        return self._xyToCell(x, y)

    def _xyToCell(self, x, y):
        return (y -1) * self.gridWidth + x

    def _cellToXY(self, cell):
        cell = cell
        y = int((cell - 1) / self.gridWidth + 1)
        x = int(cell - (y - 1) * self.gridWidth)
        return [x,y]

    def _cellToLatLon(self, cell):
        cell = cell
        xy = self._cellToXY(cell)
        xy[0] -= .5
        xy[1] -= .5
        lat = (self.gridHeight - xy[1]) * self.cellSize + self.lowerLeftY
        lon = xy[0] * self.cellSize + self.lowerLeftX
        return lat, lon


    def _latlonToXY(self, lat, lon):
        y = self.gridHeight - (int((lat - self.lowerLeftY) / self.cellSize) + 1)
+ 1
        x = int((lon - self.lowerLeftX) / self.cellSize) + 1
        return [x,y]

    def _getDistance(self, cell1, cell2):
        lat1, lon1 = self._cellToLatLon(cell1)
        lat2, lon2 = self._cellToLatLon(cell2)
        point1 = Point(lat1, lon1)
        point2 = Point(lat2, lon2)
        dist = distance(point1, point2).kilometers
        return dist


    def _xyToLatLon(self, x, y):
        cell = self._xyToCell(x, y)
        return self._cellToLatLon(cell)

    def getVertices(self):
        return self.gridVertices

    def getEdgesList(self):
        return self.gridCostList

    def getEdegsDict(self):
        return self.gridcost

    def processGeoCost(self):
        start_time = time.time()
        print("Loading geo grid...")
```

```
        self._loadgeogrid()
        print("loaded geogrid. Time Elapsed: %s seconds" %(time.time() -
start_time))
        print("")
        print("Subsetting Cost grid...")
        self._subsetGrid()
        print("Subsetting cost grid completed. Time Elapsed: %s seconds"
%(time.time() - start_time))
        print("")
        print("Loading cost...")
        self._loadcost()
        print("loaded cost. Time Elapsed: %s seconds" %(time.time() -
start_time))
        print("")



    def _subsetGrid(self):
        # nrows = self.gridHeight
        ncols = self.gridWidth

        sw = self._latlonToCell(self.south, self.west)
        se = self._latlonToCell(self.south, self.east)
        nw = self._latlonToCell(self.north, self.west)
        ne = self._latlonToCell(self.north, self.east)

        inputdata = [sw, se, nw, ne]


        newWidth = max((inputdata[1] - inputdata[0]), (inputdata[3] -
inputdata[2]))+1
        newHeight = max(abs(inputdata[2] - inputdata[0]), abs(inputdata[3] -
inputdata[1]))+ncols


        start = inputdata[0]

        n_nrows = round(newHeight/ncols)



        self.leftbounds = []
        self.rightbounds = []
        for i in range(n_nrows):
            start_x = start - (i*ncols)
            self.leftbounds.append(start_x)
            self.rightbounds.append(start_x + newWidth - 1)

        self.leftbounds.reverse()
        self.rightbounds.reverse()
```

```
    def _checkBound(self, data):
        n = len(self.leftbounds)
        left_idx = bisect_right(self.leftbounds, data[0]) - 1
        right_idx = bisect_left(self.rightbounds, data[1])

        validLeft = (left_idx >= 0 and self.leftbounds[left_idx] <= data[0] <=
self.rightbounds[left_idx])
        validRight = (right_idx < n and self.leftbounds[right_idx] <= data[1] <=
self.rightbounds[right_idx])

        valid = validLeft and validRight
        return valid

    def getHeight(self):
        return self.gridHeight

    def getWidth(self):
        return self.gridWidth

    def getCellSize(self):
        return self.cellSize
```

alternateNetworkGeo.py

```
import time
import pandas as pd
import numpy as np
from dummyCostSurface import dummyCostSurface
from networkDelanunay import networkDelanunay
from geotransformation import geoTransformation
from networkx import DiGraph
import networkx as nx
from matplotlib import rcParams
import matplotlib.pyplot as plt
from itertools import combinations
import plotly.express as px
import plotly.graph_objects as go
import random


rcParams['figure.figsize'] = 10, 8

class alternateNetworkGeo(DiGraph):
    def __init__(self, width=100, height=100):
        super().__init__()
        self.width = width
```

```
        self.height = height
        self.existingPath = {}
        self.existingPathVertices = {}
        self.existingPathType = {}
        self.sources = {}
        self.sinks = {}
        self.spathsCost = {}
        self.spaths = {}
        self.assetsXY = {}
        self.assetsLatLon = {}
        self.assetsPT = {}
        self.assetNameFromPT = {}
        self.assetNameFromXY = {}
        self.initial_pipe_spaths = {}
        self.assetCap = {}
        self.existingPathBounds = {}
        self.spathsLength = {}
        self.spathsWeight = {}



    def initialize_dummy_cost_surface(self):
        C = dummyCostSurface(width=self.width, height=self.height, lowcost=1,
highcost=60, ctype='float')
        C.generate_cost_surface()

        self.add_nodes_from(C.get_vertices())
        self.add_edges_from(C.get_ebunch())

    def initialize_cost_surface(self):
        self.gt = geoTransformation()
        self.gt.processGeoCost()

        self.width = self.gt.getWidth()
        self.height = self.gt.getHeight()

        edges = self.gt.getEdegsDict()

        cellsize = self.gt.getCellSize()


        start_time = time.time()
        print("Adding graph vertices...")
        self.add_nodes_from(self.gt.getVertices())
        print("Added Vertices. Time Taken: %s seconds" %(time.time() -
start_time))
        print("")

        # self.add_edges_from(self.gt.getEdgesList())

        start_time = time.time()
        print("Adding graph Edges...")
```

```
        diagonal_l = np.sqrt(2)*(cellsize/0.008333)
        for key in edges.keys():
            # print(key[0], key[1], edges[key])
            if (abs(key[0] - key[1]) == self.width+1) or (abs(key[0] - key[1]) ==
self.width-1):
                # approx_l = self.gt._getDistance(key[0], key[1]) #km only cal
for diagonals TODO: Too time consurming to compute actual diagonal distance
                approx_l = diagonal_l
            else:
                approx_l = cellsize/0.008333 #km
            self.add_edge(key[0], key[1], weight=edges[key], length=approx_l)
        print("Added Edges. Time Taken: %s seconds" %(time.time() - start_time))
        print("")


    def add_vertices_from_list(self, vertices):
        self.add_nodes_from(vertices)

    def add_edges_from_list(self, edgelist):
        self.add_edges_from(edgelist)


    def import_pipeline(self, input_dir, pathname, flowtype='bidirectional'):
        pipeline = pd.read_excel(input_dir)
        pipe_nodes = []
        start_nodes = pipeline['Start'].values
        end_nodes = pipeline['End'].values
        lower_bound = pipeline['Lower Cap'].values[0]
        upper_bound = pipeline['Upper Cap'].values[0]

        for i in range(len(pipeline)):
            pipe_nodes.append((start_nodes[i], end_nodes[i]))

        self.add_existing_zero_cost_path(pathname, pipe_nodes, flowtype)
        self.existingPathType[pathname] = flowtype
        self.existingPathBounds[pathname] = [lower_bound, upper_bound]


    def import_pipeline_lat_long(self, input_dir, flowtype='bidirectional'):
        print("Importing Pipeline...")
        pipeline = pd.read_excel(input_dir)
        pipe_nodes = []
        start_nodes = []
        end_nodes = []

        pathname = pipeline["Name"][0]
        lower_bound = pipeline['Lower Cap'].values[0]
        upper_bound = pipeline['Upper Cap'].values[0]

        for i in range(len(pipeline)):
            cell = self.gt._latlonToCell(pipeline["Lat"][i], pipeline["Long"][i])
            if i == 0:
```

```
                start_nodes.append(cell)
            elif i == len(pipeline)-1:
                end_nodes.append(cell)
            else:
                start_nodes.append(cell)
                end_nodes.append(cell)

        for i in range(len(start_nodes)):
            if start_nodes[i] != end_nodes[i]:
                pipe_nodes.append((start_nodes[i], end_nodes[i]))

        # print("PIPE NODES")
        # print(pipe_nodes)

        edges = [edge for edge in self.edges]
        # print(edges)
        pipe_nodes_mod = []
        for nodepair in pipe_nodes:
            if nodepair in edges:
                pipe_nodes_mod.append(nodepair)
            else:
                start_list = []
                end_list = []
                s_p = nx.shortest_path(self, nodepair[0], nodepair[1],
weight='weight')
                for i in range(len(s_p)):
                    if i == 0:
                        start_list.append(s_p[i])
                    elif i == len(s_p) - 1:
                        end_list.append(s_p[i])
                    else:
                        start_list.append(s_p[i])
                        end_list.append(s_p[i])

                for i in range(len(start_list)):
                    pipe_nodes_mod.append((start_list[i], end_list[i]))

        # print("PIPE NODE MOD: ", pipe_nodes_mod)

        print("Embedding zero cost path...")
        self.add_existing_zero_cost_path(pathname, pipe_nodes_mod, flowtype)
        self.existingPathType[pathname] = flowtype
        self.existingPathBounds[pathname] = [lower_bound, upper_bound]
        print("Finished Adding Pipeline.")
        print("")




    def add_existing_zero_cost_path(self, pathname, path_nodes, flowtype):
```

```
        existingPathVertices = {}
        np = 0
        edges = [edge for edge in self.edges]
        for nodepair in path_nodes:
            # self.edges[nodepair[0], nodepair[1]]['weight'] = 0
            if flowtype == 'bidirectional':
                if (nodepair[0], nodepair[1]) in edges:
                    self.edges[nodepair[0], nodepair[1]]['weight'] = 0
                else:
                    self.add_edge(nodepair[0], nodepair[1], weight=0)

                if (nodepair[1], nodepair[0]) in edges:
                    self.edges[nodepair[1], nodepair[0]]['weight'] = 0
                else:
                    self.add_edge(nodepair[1], nodepair[0], weight=0)
            elif flowtype == 'unidirectional':
                if (nodepair[0], nodepair[1]) in edges:
                    self.edges[nodepair[0], nodepair[1]]['weight'] = 0
                else:
                    self.add_edge(nodepair[0], nodepair[1], weight=0)

                if (nodepair[1], nodepair[0]) in edges:
                    self.edges[nodepair[1], nodepair[0]]['weight'] = 1e9
                else:
                    self.add_edge(nodepair[1], nodepair[0], weight=1e9)

            if pathname in self.existingPath:
                self.existingPath[pathname].append(nodepair)
            else:
                self.existingPath[pathname] = [nodepair]

            if pathname in existingPathVertices:
                existingPathVertices[pathname].append(nodepair[0])
            else:
                existingPathVertices[pathname] = [nodepair[0]]

            np = nodepair[1]

        existingPathVertices[pathname].append(np)
        self.existingPathVertices = existingPathVertices

    def get_existing_zero_cost_path(self):
        return self.existingPath

    def get_existing_zero_cost_path_vertices(self):
        return self.existingPathVertices

    def get_initial_pipe_spaths(self):
        return self.initial_pipe_spaths

    def enforce_pipeline_tie_point(self, pathname=None, point1=None, point2=None,
exclusion=False, etype='before', onlyin=False, onlyout=False):
```

```
        #convert x,y to points on the graph
        # if point1:
        #     point1 = self.gt._xyToCell(point1[0], point1[1])
        # if point2:
        #     point2 = self.gt._xyToCell(point2[0], point2[1])
        print("Enforcing Pipeline Tie-in Points")
        if point1:
            point1 = self.gt._latlonToCell(float(point1[0]), float(point1[1]))
        if point2:
            point2 = self.gt._latlonToCell(float(point2[0]), float(point2[1]))

        # print("POINT1 & 2: ", point1, point2)

        if pathname is None:
            keys = [key for key in self.existingPath.keys()]
            pathname = keys[0]


        #case 1: 2 tie in points with all exclusion
        if point1 and point2 and (not exclusion):
            print("case 1: 2 tie in points with all exclusion")
            for edge in self.edges:
                #in
                if (edge[1] in self.existingPathVertices[pathname]) and (edge[0]
not in self.existingPathVertices[pathname]) \
                    and (edge[1] != point1) and (edge[1] != point2):
                    self.edges[edge]['weight'] = 1e9

                #out
                if (edge[0] in self.existingPathVertices[pathname]) and (edge[1]
not in self.existingPathVertices[pathname]) \
                    and (edge[0] != point1) and (edge[0] != point2):
                    self.edges[edge]['weight'] = 1e9

                if onlyin:
                    if ((edge[0] == point1) or (edge[0] == point2)) and (edge[1]
not in self.existingPathVertices[pathname]):
                        self.edges[edge]['weight'] = 1e9

                if onlyout:
                    if ((edge[1] == point1) or (edge[1] == point2)) and (edge[0]
not in self.existingPathVertices[pathname]):
                        self.edges[edge]['weight'] = 1e9


        #case 2: 2 tie in points with exclusion at ends
        elif point1 and point2 and exclusion:
            print("#case 2: 2 tie in points with exclusion at ends")
            #get all the vertices before and after point 1 and 2
            pathvertices = self.existingPathVertices[pathname].copy()
            minidx, maxidx = map(pathvertices.index, (point1, point2))
            minidx, maxidx = min(minidx, maxidx), max(minidx, maxidx)
```

```
            not_excluded = pathvertices[minidx:maxidx+1]
            exclusion_list = [i for i in pathvertices if i not in not_excluded]

            for edge in self.edges:
                #in
                if (edge[1] in exclusion_list) and (edge[0] not in pathvertices)
\
                    and (edge[1] != point1) and (edge[1] != point2):
                    self.edges[edge]['weight'] = 1e9

                #out
                if (edge[0] in exclusion_list) and (edge[1] not in pathvertices)
\
                    and (edge[0] != point1) and (edge[0] != point2):
                    self.edges[edge]['weight'] = 1e9

                if onlyin:
                    if (edge[0] in not_excluded) and (edge[1] not in
self.existingPathVertices[pathname]):
                        self.edges[edge]['weight'] = 1e9

                if onlyout:
                    if (edge[1] in not_excluded) and (edge[0] not in
self.existingPathVertices[pathname]):
                        self.edges[edge]['weight'] = 1e9


        else:
            #case 3 single point with all exclusion but source/sink
            if (point1 or point2) and (not exclusion):
                print("#case 3 single point with all exclusion but source/sink")
                point = point1 or point2
                pathvertices = self.existingPathVertices[pathname].copy()
                if pathvertices[0] > pathvertices[-1]: #make the path always read
from left to right
                    pathvertices.reverse()
                if etype == 'before':
                    exclusion_list = pathvertices[:-1] #means source/sink is at
end of path
                else:
                    exclusion_list = pathvertices[1:] #means source/sink is at
begining of path

                for edge in self.edges:
                    #in
                    if (edge[1] in exclusion_list) and (edge[0] not in
pathvertices) \
                        and (edge[1] != point):
                        self.edges[edge]['weight'] = 1e9
                    #out
                    if (edge[0] in exclusion_list) and (edge[1] not in
pathvertices) \
```

```
                        and (edge[0] != point):
                        self.edges[edge]['weight'] = 1e9

                    #enforce onlyin
                    if onlyin:
                        if (edge[0] == point) and (edge[1] not in pathvertices):
                            self.edges[edge]['weight'] = 1e9

                    #enforce onlyin
                    if onlyout:
                        if (edge[1] == point) and (edge[0] not in pathvertices):
                            self.edges[edge]['weight'] = 1e9



            #case 4 single point with before or after exclusion on one end
            elif (point1 or point2) and exclusion:
                print("#case 4 single point with before or after exclusion on one
end")
                point = point1 or point2
                pathvertices = self.existingPathVertices[pathname].copy()
                if pathvertices[0] > pathvertices[-1]: #make the path always read
from left to right
                    pathvertices.reverse()
                if etype == 'before':
                    exclusion_list = pathvertices[:pathvertices.index(point)]
                else:
                    exclusion_list = pathvertices[pathvertices.index(point)+1:]

                end1 = pathvertices[0]
                end2 = pathvertices[-1]

                for edge in self.edges:
                    #in
                    if (edge[1] in exclusion_list) and (edge[0] not in
pathvertices) \
                        and (edge[1] != point):
                        self.edges[edge]['weight'] = 1e9

                    #out
                    if (edge[0] in exclusion_list) and (edge[1] not in
pathvertices) \
                        and (edge[0] != point):
                        self.edges[edge]['weight'] = 1e9


                    if onlyin:
                        if (edge[0] in pathvertices) and (edge[0] not in
exclusion_list) \
                            and (edge[0] != end2) and (edge[0] != end1) and
(edge[1] not in pathvertices):
                            self.edges[edge]['weight'] = 1e9
```

```
                    if onlyout:
                        if (edge[1] in pathvertices) and (edge[1] not in
exclusion_list) \
                            and (edge[1] != end2) and (edge[1] != end1) and
(edge[0] not in pathvertices):
                            self.edges[edge]['weight'] = 1e9
        print("")



    def enforce_no_pipeline_diagonal_Xover(self):
        print("Enforcing no diagonal pipeline crossing...")
        edges = [edge for edge in self.edges]

        for pathname in self.existingPath.keys():
            for nodepair in self.existingPath[pathname]:
                if abs(nodepair[0] - nodepair[1]) == self.width+2:
                    lower_diag = min(nodepair) + 1
                    upper_diag = max(nodepair) - 1

                    if (lower_diag, upper_diag) in edges:
                        self.edges[(lower_diag, upper_diag)]['weight'] = 1e9
                    if (upper_diag, lower_diag) in edges:
                        self.edges[(upper_diag, lower_diag)]['weight'] = 1e9

                elif abs(nodepair[0] - nodepair[1]) == self.width:
                    lower_diag = min(nodepair) - 1
                    upper_diag = max(nodepair) + 1

                    if (lower_diag, upper_diag) in edges:
                        self.edges[(lower_diag, upper_diag)]['weight'] = 1e9
                    if (upper_diag, lower_diag) in edges:
                        self.edges[(upper_diag, lower_diag)]['weight'] = 1e9
        print('No pipeline diaginal crossing enforced')
        print("")
        return

    def enforce_no_path_diagonal_Xover(self, path_tup):
        print("Enforcing no diagonal path crossing...")
        edges = [edge for edge in self.edges]

        for nodepair in path_tup:
            if abs(nodepair[0] - nodepair[1]) == self.width+2:
                lower_diag = min(nodepair) + 1
                upper_diag = max(nodepair) - 1

                if (lower_diag, upper_diag) in edges:
                    self.edges[(lower_diag, upper_diag)]['weight'] = 1e9
                if (upper_diag, lower_diag) in edges:
                    self.edges[(upper_diag, lower_diag)]['weight'] = 1e9
```

```
            elif abs(nodepair[0] - nodepair[1]) == self.width:
                lower_diag = min(nodepair) - 1
                upper_diag = max(nodepair) + 1

                if (lower_diag, upper_diag) in edges:
                    self.edges[(lower_diag, upper_diag)]['weight'] = 1e9
                if (upper_diag, lower_diag) in edges:
                    self.edges[(upper_diag, lower_diag)]['weight'] = 1e9
        print('No pipeline diaginal crossing enforced')
        print("")
        return


    def add_sources(self, sourcelist):
        for source in sourcelist:
            self.sources[source[0]] = [source[1], source[2]]
            self.assetsLatLon[source[0]] = [source[1], source[2]]
            xy = self.gt._latlonToXY(source[1], source[2])
            self.assetsXY[source[0]] = [xy[0], xy[1]]
            self.assetCap[source[0]] = source[3]

    def add_sinks(self, sinklist):
        for sink in sinklist:
            self.sinks[sink[0]] = [sink[1], sink[2]]
            self.assetsLatLon[sink[0]] = [sink[1], sink[2]]
            xy = self.gt._latlonToXY(sink[1], sink[2])
            self.assetsXY[sink[0]] = [xy[0], xy[1]]
            self.assetCap[sink[0]] = -sink[3]

    def _generate_assetsPT(self):
        for key in self.assetsXY.keys():
            self.assetsPT[key] = self.gt._xyToCell(self.assetsXY[key][0],
self.assetsXY[key][1])

        for key in self.assetsPT.keys():
            self.assetNameFromPT[self.assetsPT[key]] = key

        for key in self.assetsXY.keys():
            self.assetNameFromXY[(self.assetsXY[key][0], self.assetsXY[key][1])]
= key



    def generateDelaunayNetwork(self):
        print("Generating Delanuay Network...")
        self.D = networkDelanunay(width=self.width, height=self.height)
        assets = []
        for key, asset in self.assetsXY.items():
            assets.append(asset)

        assets = np.array(assets)
        self.D.add_points_from_list(assets)
```

```
        self.D.createDelaunayNetwork()
        print("Delaunay network generated")
        print('')


    def showDelaunayNetwork(self):
        self.D.plotNetwork()

    def add_Delaunay_tiepoints(self, tieptslist):
        for tiepts in tieptslist:
            if tiepts[0] in self.sources.keys():
                node1 = self.sources[tiepts[0]]
            elif tiepts[0] in self.sinks.keys():
                node1 = self.sinks[tiepts[0]]

            if tiepts[2] in self.sources.keys():
                node2 = self.sources[tiepts[2]]
            elif tiepts[2] in self.sinks.keys():
                node2 = self.sinks[tiepts[2]]

            self.assetsXY[f"Tnode from {tiepts[0]} to pipeline1"] = tiepts[1]
            self.assetsXY[f"Tnode from {tiepts[2]} to pipeline1"] = tiepts[3]


            self.D.add_tie_in_point(node1,tiepts[1])
            self.D.add_tie_in_point(node2, tiepts[3])
            self.D.add_tie_in_point(tiepts[1], tiepts[3])
            self.D.delete_line_path(node1, node2)

            self.enforce_pipeline_tie_point(tiepts[4], tiepts[1], tiepts[3])

        return

    def get_sources(self):
        return self.sources

    def get_sinks(self):
        return self.sinks

    def print_edges(self):
        for edge in self.edges:
            print(edge, self.edges[edge])

    def weight_func(self, distance, time):
        return distance * time

    def get_shortest_path_and_length(self, source, destination):
        # slength = nx.shortest_path_length(self, source, destination,
weight=lambda u, v, d: self.weight_func(d['weight'], d['length']))
        # spath = nx.shortest_path(self, source, destination, weight=lambda u, v,
d: self.weight_func(d['weight'], d['length']))
```

```
        slength = nx.shortest_path_length(self, source, destination,
weight='weight')
        spath = nx.shortest_path(self, source, destination, weight='weight')
        return slength, spath

    def get_all_source_sink_shortest_paths(self):
        print('Generating all Delaunay pair shortest path...')
        self.lines = self.D.getDelaunayNetwork()
        for line in self.lines:
            cost, path = self.get_shortest_path_and_length(line[0], line[1])
            self.spathsCost[(line[0], line[1])] = cost
            self.spaths[(line[0], line[1])] = path
            self.initial_pipe_spaths[(line[0], line[1])] = path

            path_tup = [(path[i], path[i+1]) for i in range(len(path)-1)]
            # self.enforce_no_path_diagonal_Xover(path_tup)

        self._generate_assetsPT()
        print('Done generating shortest paths.')
        print("")

    def get_spathsCost(self):
        return self.spathsCost

    def print_candidate_shortest_paths(self):
        print("The lengths are: ", self.spathsLength)
        print("")
        print("The weights are: ", self.spathsWeight)
        print("")
        print("The weighted costs are: ", self.spathsCost)
        print("")
        print("The shortest paths are: ", self.spaths)
        print("")
        print(len(self.spathsCost), len(self.spaths))

    def print_assets(self):
        print(self.assetsPT)
        print(self.assetsXY)
        print(len(self.assetsPT), len(self.assetsXY))

    def show_candidate_network(self):
        rcParams['figure.figsize'] = 20, 20

        # ptslist = self.nodes
        ptslist = self.edges

        self._generate_assetsPT()


        #plot the shortest paths between nodes
        for key in self.spaths.keys():
            xs = []
```

```
            ys = []
            for pt in self.spaths[key]:
                xy = self.gt._cellToXY(pt)
                xs.append(xy[0])
                ys.append(xy[1])
            plt.plot(xs, ys, label=f"path between {self.assetNameFromPT[key[0]]}
and {self.assetNameFromPT[key[1]]}", lw = 5)

        #plot all existing pipelines
        for key in self.existingPathVertices.keys():
            xp = []
            yp = []
            for pt in self.existingPathVertices[key]:
                xy = self.gt._cellToXY(pt)
                xp.append(xy[0])
                yp.append(xy[1])
            plt.plot(xp, yp, 'red', lw=6, alpha=0.5)


        #plot asset markers
        for key in self.assetsXY.keys():
            x = self.assetsXY[key][0]
            y = self.assetsXY[key][1]
            if "node" in key:
                plt.plot(x,y, marker='o', mfc='orange', ms=5, mec='black')
            elif "source" in key:
                plt.plot(x,y, marker="s", mfc='black', ms=5, mec='black')
                plt.plot(x,y, marker=f"${key}$", mfc='black', ms=40, mec='black')
            elif "sink" in key:
                plt.plot(x,y, marker="s", mfc='yellow', ms=5, mec='red')
                plt.plot(x,y, marker=f"${key}$", mfc='yellow', ms=30, mec='red')

        plt.title("Candidate CO₂ Sequestration Network")
        plt.xlabel("X location")
        plt.ylabel("Y location")
#         plt.legend()
        plt.show()

        return

    def extract_network(self):
        return

    def get_pipe_trans_nodes(self):
        print('Generating Pipeline transshipment nodes...')
        existingPathVertices = self.existingPathVertices.copy()
        spaths = self.spaths.copy()

        trans_nodes = {}
        conn_to_del = []
        for pathname in existingPathVertices.keys():
            for nodepair in spaths.keys():
```

```
                entry = False
                start = self.spaths[nodepair][0]
                end = self.spaths[nodepair][-1]
                for i in range(len(spaths[nodepair])):
                    if (entry == False) and (spaths[nodepair][i] in
existingPathVertices[pathname]):
                        entry = True
                        trans_nodes[(pathname, nodepair)] = [spaths[nodepair][i]]
                    if (entry == True) and (spaths[nodepair][i] not in
existingPathVertices[pathname]):
                        trans_nodes[(pathname,
nodepair)].append(spaths[nodepair][i-1])
                        break

                if (entry == True) and (len(trans_nodes[(pathname, nodepair)]) ==
1):
                    trans_nodes[(pathname, nodepair)].append(spaths[nodepair][-
1])

                if entry == True:
                    node1 = trans_nodes[(pathname, nodepair)][0]
                    node2 = trans_nodes[(pathname, nodepair)][1]
                    idx1 = spaths[nodepair].index(node1)
                    idx2 = spaths[nodepair].index(node2)

                    self._generate_assetsPT()


                    self.spaths[(node1, node2)] =
self.spaths[nodepair][idx1:idx2+1]
                    self.spaths[(start, node1)] = self.spaths[nodepair][0:idx1+1]
                    self.spaths[(node2, end)] = self.spaths[nodepair][idx2:]


                    cost_1 = 0
                    cost_2 = 0
                    for i in range(len(self.spaths[(start, node1)])-1):
                        cost_1 += self.edges[self.spaths[(start, node1)][i],
self.spaths[(start, node1)][i+1]]['weight']
                    for i in range(len(self.spaths[(node2, end)])-1):
                        cost_2 += self.edges[self.spaths[(node2, end)][i],
self.spaths[((node2, end))][i+1]]['weight']


                    self.spathsCost[(node1, node2)] = 0
                    self.spathsCost[(start, node1)] = cost_1
                    self.spathsCost[(node2, end)] = cost_2



                    from_name = self.assetNameFromPT[nodepair[0]]
                    to_name = self.assetNameFromPT[nodepair[1]]
```

```
                    self.assetsXY[pathname + f" from {from_name} to {to_name}
node1"] = self.gt._cellToXY(node1)
                    self.assetsXY[pathname + f" from {from_name} to {to_name}
node2"] = self.gt._cellToXY(node2)

                    conn_to_del.append((start, end))

        for conn in conn_to_del:
            del self.spaths[conn]
            del self.spathsCost[conn]

        print('Pipeline Transshipment Nodes generated.')
        print('')
        return


    def pipe_post_process(self):
        print("Post processing Pipeline Paths...")
        self._generate_assetsPT()

        for pathname in self.existingPath.keys():
            nodes_on_pipe = []
            for key in self.assetNameFromPT.keys():
                if key in self.existingPathVertices[pathname]:
                    nodes_on_pipe.append((key,
self.existingPathVertices[pathname].index(key)))
            nodes_on_pipe = list(set(nodes_on_pipe))
            nodes_on_pipe.sort(key=lambda x: x[1])
            nodes_on_pipe = [i for (i, j) in nodes_on_pipe]

            list_combinations = list()

            for n in range(len(nodes_on_pipe)+1):
                list_combinations += list((combinations(nodes_on_pipe, n)))

            list_combinations = [tup for tup in list_combinations if len(tup) ==
2]

            joints = [(nodes_on_pipe[i], nodes_on_pipe[i+1]) for i in
range(len(nodes_on_pipe)-1)]
            edges_to_remove = [tup for tup in list_combinations if tup not in
joints]

            #remove redundant edges
            for edge in edges_to_remove:
                if edge in self.spaths.keys():
                    del self.spaths[edge]
                if edge in self.spathsCost.keys():
                    del self.spathsCost[edge]
```

```
            #add edges with cost
            for edge in joints:
                slength, spath = self.get_shortest_path_and_length(edge[0],
edge[1])
                self.spaths[edge] = spath
                self.spathsCost[edge] = slength

        self._generate_assetsPT()
        print("Pipeline post process complete.")
        print('')


    def _print_assetNameFromPT(self):
        print(self.assetNameFromPT)
        print(len(self.assetNameFromPT))

    def get_trans_nodes(self):
        print('Generating paths transshipment nodes...')
        self._generate_assetsPT()
        spaths = self.spaths.copy()

        trans_nodes = {}
        conn_to_del = []

        for pathname in spaths.keys():
            for nodepair in spaths.keys():
                entry = False
                start = spaths[nodepair][0]
                end = spaths[nodepair][-1]

                for i in range(len(spaths[nodepair])):
                    if (entry == False) and (spaths[nodepair][i] in
spaths[pathname]):
                        entry = True
                        trans_nodes[(pathname, nodepair)] = [spaths[nodepair][i]]
                    if (entry == True) and (spaths[nodepair][i] not in
spaths[pathname]):
                        trans_nodes[(pathname,
nodepair)].append(spaths[nodepair][i-1])
                        break


                if entry == True:
                    if len(trans_nodes[(pathname, nodepair)]) == 1:
                        trans_nodes[(pathname,
nodepair)].append(spaths[nodepair][-1])

                    if trans_nodes[(pathname, nodepair)][0] ==
trans_nodes[(pathname, nodepair)][1]:
                        pass
                    else:
                        node1 = trans_nodes[(pathname, nodepair)][0]
```

```
                        node2 = trans_nodes[(pathname, nodepair)][1]
                        idx1 = spaths[nodepair].index(node1)
                        idx2 = spaths[nodepair].index(node2)


                        self._generate_assetsPT()


                        if start != node1:
                            self.spaths[(start, node1)] =
self.spaths[nodepair][0:idx1+1]
                            cost_1 = 0
                            for i in range(len(self.spaths[(start, node1)])-1):
                                cost_1 += \
                                self.edges[self.spaths[(start, node1)][i],
self.spaths[(start, node1)][i+1]]['weight']
                            self.spathsCost[(start, node1)] = cost_1

                        if node1 != node2:
                            self.spaths[(node1, node2)] =
self.spaths[nodepair][idx1:idx2+1]
                            cost_2 = 0
                            for i in range(len(self.spaths[(node1, node2)])-1):
                                cost_2 += \
                                self.edges[self.spaths[(node1, node2)][i],
self.spaths[((node1, node2))][i+1]]['weight']
                            self.spathsCost[(node1, node2)] = cost_2

                        if node2 != end:
                            self.spaths[(node2, end)] =
self.spaths[nodepair][idx2:]
                            cost_3 = 0
                            for i in range(len(self.spaths[(node2, end)])-1):
                                cost_3 += \
                                self.edges[self.spaths[(node2, end)][i],
self.spaths[((node2, end))][i+1]]['weight']
                            self.spathsCost[(node2, end)] = cost_3

                        from_name = self.assetNameFromPT[nodepair[0]]
                        to_name = self.assetNameFromPT[nodepair[1]]


                        if node1 in self.assetNameFromPT.keys():
                            if ('sink' not in self.assetNameFromPT[node1]) \
                                and ('source' not in
self.assetNameFromPT[node1]):
                                self.assetsXY[str(pathname) + f" from {from_name}
to {to_name} node1"] = \
                                self.gt._cellToXY(node1)
                        else:
                            self.assetsXY[str(pathname) + f" from {from_name} to
{to_name} node1"] = \
```

```
                            self.gt._cellToXY(node1)

                        if node2 in self.assetNameFromPT.keys():
                            if ('sink' not in self.assetNameFromPT[node2]) \
                                and ('source' not in
self.assetNameFromPT[node2]):
                                self.assetsXY[str(pathname) + f" from {from_name}
to {to_name} node2"] = \
                                self.gt._cellToXY(node2)
                        else:
                            self.assetsXY[str(pathname) + f" from {from_name} to
{to_name} node2"] = \
                            self.gt._cellToXY(node2)

                        conn_to_del.append((start, end))
#                         self._generate_assetsPT()

        for conn in list(set(conn_to_del)):
            del self.spaths[conn]
            del self.spathsCost[conn]

        print('pipe transshipment nodes generated.')
        print('')
        return

    def trans_node_post_process(self):
        print('Started post processing of path transshipment nodes...')
        self._generate_assetsPT()

        for pathname in self.initial_pipe_spaths.keys():
            nodes_on_pipe = []
            for key in self.assetNameFromPT.keys():
                if key in self.initial_pipe_spaths[pathname]:
                    nodes_on_pipe.append((key,
self.initial_pipe_spaths[pathname].index(key)))
            nodes_on_pipe = list(set(nodes_on_pipe))
            nodes_on_pipe.sort(key=lambda x: x[1])
            nodes_on_pipe = [i for (i, j) in nodes_on_pipe]

            list_combinations = list()

            for n in range(len(nodes_on_pipe)+1):
                list_combinations += list((combinations(nodes_on_pipe, n)))

            list_combinations = [tup for tup in list_combinations if len(tup) ==
2]

            joints = [(nodes_on_pipe[i], nodes_on_pipe[i+1]) for i in
range(len(nodes_on_pipe)-1)]
            edges_to_remove = [tup for tup in list_combinations if tup not in
joints]
```

```
            #remove redundant edges
            for edge in edges_to_remove:
                if edge in self.spaths.keys():
                    del self.spaths[edge]
                if edge in self.spathsCost.keys():
                    del self.spathsCost[edge]


            #add edges with cost
            for edge in joints:
                slength, spath = self.get_shortest_path_and_length(edge[0],
edge[1])
                self.spaths[edge] = spath
                self.spathsCost[edge] = slength

        self._generate_assetsPT()
        print('path transhipment nodes processing done.')
        print('')

    def plot_extracted_graph(self):
        res = []
        for key in self.spathsCost.keys():
            pt1 = self.gt._cellToXY(key[0])
            pt2 = self.gt._cellToXY(key[1])
            res.append((pt1, pt2))

        for line in res:
            x = []
            y = []
            x.append(line[0][0])
            y.append(line[0][1])
            x.append(line[1][0])
            y.append(line[1][1])
            plt.plot(x, y,  marker='o', mfc='green', mec='green', label=f"from
{line[0]} to {line[1]}")

        plt.title('2D visualization of extracted Graph')
        plt.xlabel("X location")
        plt.ylabel('Y location')
        # plt.legend()
        plt.show()

    def shortest_paths_post_process(self):
        print('Post processing of shortest paths initiated...')
        spaths = self.spaths.copy()
        spathsCost = self.spathsCost.copy()

        for key in self.spaths.keys():
            if ((key[0], key[1]) in spaths.keys()) and ((key[1], key[0]) in
spaths.keys()):
                del spaths[(key[1], key[0])]
                del spathsCost[(key[1], key[0])]
```

```
    self.spaths = spaths.copy()
    self.spathsCost = spathsCost.copy()

    for key in spaths.keys():
        t_length = 0
        t_weight = 0
        for i in range(len(spaths[key]) - 1):
            nodelist = spaths[key]
            t_weight += self.edges[(nodelist[i], nodelist[i+1])]['weight']
            t_length += self.edges[(nodelist[i], nodelist[i+1])]['length']
        self.spathsWeight[key] = t_weight
        self.spathsLength[key] = t_length

    print('shortest paths post processing completed.')
    print('')


def _getMappingData(self):
    print('Generating Mapping Data...')
    assets_df = {"Name": [],
            "Lat": [],
            "Lon": [],
            "Type": []}

    for key in self.assetsXY.keys():
        x = self.assetsXY[key][0]
        y = self.assetsXY[key][1]
        lat, lon = self.gt._xyToLatLon(x, y)
        assets_df["Name"].append(key)
        assets_df["Lat"].append(lat)
        assets_df["Lon"].append(lon)
        if "node" in key:
            assets_df["Type"].append("node")
        elif "source" in key:
            assets_df["Type"].append("source")
        elif "sink" in key:
            assets_df["Type"].append("sink")


    self.assets_df = pd.DataFrame(assets_df)

    pipelines_df = {"Name": [],
                    "Lat": [],
                    "Lon": []}

    for key in self.spaths.keys():
        for pt in self.spaths[key]:
            lat, lon = self.gt._cellToLatLon(pt)
            pipelines_df["Name"].append(key)
            pipelines_df["Lat"].append(lat)
            pipelines_df["Lon"].append(lon)
```

```
        self.pipelines_df = pd.DataFrame(pipelines_df)

        self.unique_pipes = self.pipelines_df["Name"].unique()


        existing_path_df = {"Name": [],
                        "Lat": [],
                        "Lon": []}

        for key in self.existingPathVertices.keys():
            for pt in self.existingPathVertices[key]:
                lat, lon = self.gt._cellToLatLon(pt)
                existing_path_df["Name"].append(key)
                existing_path_df["Lat"].append(lat)
                existing_path_df["Lon"].append(lon)

        self.existing_path_df = pd.DataFrame(existing_path_df)

        # self.lines = self.D.getDelaunayNetwork()
        print('Mapping Data sucessfully generated.')
        print('')


    def _getDelaunayMapFig(self):
        assets_subset = self.assets_df[self.assets_df["Type"].isin(["source",
"sink"])]
        fig1 = px.scatter_mapbox(assets_subset, lat="Lat", lon="Lon",
hover_name="Name", color="Type", zoom=7, height=1000, width=1000, size="Lat",
                                 color_discrete_map={"source":"red",
"sink":"green"})
        fig1.update_layout(mapbox_style="open-street-map")


        for line in self.lines:
            lat1, lon1 = self.gt._cellToLatLon(line[0])
            lat2, lon2 = self.gt._cellToLatLon(line[1])
            fig1.add_trace(go.Scattermapbox(
                mode = "lines",
                lat = [lat1, lat2],
                lon = [lon1, lon2],
                showlegend=False,
                line={'color':'black'}
            ))

        return fig1

    def _getAlternateNetworkMapFig(self):
        fig2 = px.scatter_mapbox(self.assets_df, lat="Lat", lon="Lon",
hover_name="Name", color="Type", zoom=7, height=1000, width=1000, size="Lat",
                                 color_discrete_map={"source":"red",
"sink":"green",  "node":"orange"})
```

```
        fig2.update_layout(mapbox_style="open-street-map")

        for pipe in self.unique_pipes:
            fig2.add_trace(go.Scattermapbox(
                mode = "lines",
                lat = self.pipelines_df[self.pipelines_df.Name == pipe]["Lat"],
                lon = self.pipelines_df[self.pipelines_df.Name == pipe]["Lon"],
                showlegend=False,
                line={'color':'blue'},
                name = str(pipe)
            ))

        #addpipeline plot
        for path in self.existingPath.keys():
            fig2.add_trace(go.Scattermapbox(
                mode = "lines",
                lat = self.existing_path_df[self.existing_path_df.Name ==
path]["Lat"],
                lon = self.existing_path_df[self.existing_path_df.Name ==
path]["Lon"],
                showlegend=True,
                opacity=0.5,
                line={'width': 5, 'color':'purple'},
                name = str(path)
            ))


        return fig2

    def _getSolnNetworkMapFig(self, soln_arcs, point1=None, point2=None,
show_alt=True):
        fig3 = px.scatter_mapbox(self.assets_df, lat="Lat", lon="Lon",
hover_name="Name", color="Type", zoom=7, height=1000, width=1000, size="Lat",
                                 color_discrete_map={"source":"red",
"sink":"green",  "node":"orange"})
        fig3.update_layout(mapbox_style="open-street-map")

        if show_alt:
            for pipe in self.unique_pipes:
                fig3.add_trace(go.Scattermapbox(
                    mode = "lines",
                    lat = self.pipelines_df[self.pipelines_df.Name ==
pipe]["Lat"],
                    lon = self.pipelines_df[self.pipelines_df.Name ==
pipe]["Lon"],
                    showlegend=False,
                    line={'color':'blue'},
                    name = str(pipe)
                ))

        if point1:
            if point1 != ["", ""]:
```

```
                fig3.add_trace(go.Scattermapbox(
                        mode = "markers",
                        lat = [point1[0]],
                        lon = [point1[1]],
                        showlegend=True,
                        opacity=0.8,
                        marker={'size': 20, 'color':'orange'},
                        name = str("Tie-in Point 1")
                        ))
                fig3.add_trace(go.Scattermapbox(
                mode = "markers",
                lat = [point1[0]],
                lon = [point1[1]],
                showlegend=False,
                opacity=0.8,
                marker={'size': 10, 'color':'black'},
                name = str("Tie-in Point 1")
                ))

        if point2:
            if point2 != ["", ""]:
                fig3.add_trace(go.Scattermapbox(
                        mode = "markers",
                        lat = [point2[0]],
                        lon = [point2[1]],
                        showlegend=True,
                        opacity=0.8,
                        marker={'size': 20, 'color':'orange'},
                        name = str("Tie-in Point 2")
                        ))
                fig3.add_trace(go.Scattermapbox(
                mode = "markers",
                lat = [point2[0]],
                lon = [point2[1]],
                showlegend=False,
                opacity=0.8,
                marker={'size': 10, 'color':'black'},
                name = str("Tie-in Point 2")
                ))

        #addpipeline plot
        for path in self.existingPath.keys():
            fig3.add_trace(go.Scattermapbox(
                mode = "lines",
                lat = self.existing_path_df[self.existing_path_df.Name ==
path]["Lat"],
                lon = self.existing_path_df[self.existing_path_df.Name ==
path]["Lon"],
                showlegend=True,
                opacity=0.5,
                line={'width': 5, 'color':'purple'},
                name = str(path)
```

```
            ))

        #highlight soln
        for pipe in self.unique_pipes:
            if ((self.nodesdict[pipe[0]], self.nodesdict[pipe[1]]) in
soln_arcs.keys()) or \
                ((self.nodesdict[pipe[1]], self.nodesdict[pipe[0]]) in
soln_arcs.keys()) :
                fig3.add_trace(go.Scattermapbox(
                    mode = "lines",
                    lat = self.pipelines_df[self.pipelines_df.Name ==
pipe]["Lat"],
                    lon = self.pipelines_df[self.pipelines_df.Name ==
pipe]["Lon"],
                    showlegend=False,
                    opacity=0.5,
                    line={'width': 10, 'color':'green'}

                ))

        return fig3


    def _getSolnResults(self, soln_arcs):
        solnkeys = []
        for pipe in self.unique_pipes:
            if ((self.nodesdict[pipe[0]], self.nodesdict[pipe[1]]) in
soln_arcs.keys()):
                solnkeys.append((pipe[0], pipe[1], 'n'))
            elif ((self.nodesdict[pipe[1]], self.nodesdict[pipe[0]]) in
soln_arcs.keys()):
                solnkeys.append((pipe[0], pipe[1], 'r'))

        resultdict2 = {}
        for solnkey in solnkeys:
            path = self.spaths[(solnkey[0], solnkey[1])]
            path_geo = []
            length_act = 0
            for i in range(len(path)-1):
                lat1, lon1 = self.gt._cellToLatLon(path[i])
                lat2, lon2 = self.gt._cellToLatLon(path[i+1])
                path_geo.append(((lat1, lon1), (lat2, lon2)))
                length_act += self.gt._getDistance(path[i], path[i+1])
            length1 = self.spathsLength[(solnkey[0], solnkey[1])]
            length = length_act
            if solnkey[2] == 'n':
                resultdict2[(self.nodesdict[solnkey[0]],
self.nodesdict[solnkey[1]])] = {"length": length, "path": path_geo}
            else:
                path_geo.reverse()
                resultdict2[(self.nodesdict[solnkey[1]],
self.nodesdict[solnkey[0]])] = {"length": length, "path": path_geo}
```

```
    return resultdict2



def export_network(self):
    self.nodesdict = {}
    nodenames = []
    idx = 1
    pipe_idx = 1
    for key,value in self.assetNameFromPT.items():
        for pipeline in self.existingPath.keys():
            if pipeline in value:
                self.nodesdict[key] = f'{pipeline}_TS'+str(pipe_idx)
                nodenames.append(f'{pipeline}_TS'+str(pipe_idx))
                pipe_idx+=1
                break
        if ('from' in value) and (key not in self.nodesdict.keys()):
            self.nodesdict[key] = 'TS'+str(idx)
            nodenames.append('TS'+str(idx))
            idx+=1
        elif key not in self.nodesdict.keys():
            self.nodesdict[key] = value
            nodenames.append(value)

    # print("nodesdict: ", nodesdict)
    arcsCost = {}
    arcsLength = {}
    arcsWeight = {}
    arcsPath = {}
    arcs = []


    for key, value in self.spathsCost.items():
        node1 = self.nodesdict[key[0]]
        node2 = self.nodesdict[key[1]]
        arc_1 = node1.split("_")[0]
        arc_2 = node2.split("_")[0]
        if (arc_1 == arc_2) and (arc_1 in self.existingPathBounds.keys()):
            if (self.existingPathType[arc_1] == "unidirectional"):
                arcsCost[(node1, node2)] = value
                arcsLength[(node1, node2)] = self.spathsLength[key]
                arcsWeight[(node1, node2)] = self.spathsWeight[key]
                arcsPath[(node1, node2)] = self.spaths[key]
                arcs.append((node1, node2))
            else:
                arcsCost[(node1, node2)] = value
                arcsCost[(node2, node1)] = value
                arcsLength[(node1, node2)] = self.spathsLength[key]
                arcsLength[(node2, node1)] = self.spathsLength[key]
                arcsWeight[(node1, node2)] = self.spathsWeight[key]
                arcsWeight[(node2, node1)] = self.spathsWeight[key]
```

```
                    arcsPath[(node1, node2)] = self.spaths[key]
                    arcsPath[(node2, node1)] = [i for i in
reversed(self.spaths[key])]
                    arcs.append((node1, node2))
                    arcs.append((node2, node1))
            else:
                arcsCost[(node1, node2)] = value
                arcsCost[(node2, node1)] = value
                arcsLength[(node1, node2)] = self.spathsLength[key]
                arcsLength[(node2, node1)] = self.spathsLength[key]
                arcsWeight[(node1, node2)] = self.spathsWeight[key]
                arcsWeight[(node2, node1)] = self.spathsWeight[key]
                arcsPath[(node1, node2)] = self.spaths[key]
                arcsPath[(node2, node1)] = [i for i in
reversed(self.spaths[key])]
                arcs.append((node1, node2))
                arcs.append((node2, node1))


        #get b values for network graph
        nodes_b ={key:0 for key in nodenames}

        for node in nodes_b:
            if node in self.assetCap.keys():
                nodes_b[node] = self.assetCap[node]

        #define all arc info [length, weight, w_cost, lower_bound, upper_bound]
        arcsInfo = {key:[arcsLength[key],arcsWeight[key],arcsCost[key], 0, 1e9]
for key in arcsCost.keys()}

        for arc in arcs:
            arc_1 = arc[0].split("_")[0]
            arc_2 = arc[1].split("_")[0]
            if arc_1 == arc_2:
                if arc_1 in self.existingPathBounds.keys():
                    arcsInfo[arc][3] = self.existingPathBounds[arc_1][0]
                    arcsInfo[arc][4] = self.existingPathBounds[arc_1][1]

        return nodenames, arcs, arcsInfo, arcsPath, nodes_b
```

math_model.py

```
import os
import pulp as pl
from pulp import *
from typing import Dict, List, Set
import pandas as pd
import numpy as np
```

```
import gurobipy as gp
import logging
import os
from gurobipy import GRB
from alternateNetworkGeo import alternateNetworkGeo
import time
import datetime


MPS_FILE_PATH =
os.path.join("Sequestrix/app/solver_files/CO₂_network_optimization.mps")
LP_FILE_PATH =
os.path.join("Sequestrix/app/solver_files/CO₂_network_optimization.lp")
SOL_FILE_PATH =
os.path.join("Sequestrix/app/solver_files/CO₂_network_optimization.sol")
ILP_FILE_PATH =
os.path.join("Sequestrix/app/solver_files/CO₂_network_optimization.ilp")

LOGGER = logging.getLogger(__name__)
FORMAT = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
logging.basicConfig(filename='model_solve.log', filemode='w',
level=logging.DEBUG, format=FORMAT)
START_TIME = time.time()

class Math_model:
    def __init__(self, nodes, nodesValue, arcs, arcsInfo, paths, nodesCost,
duration, target_cap, crf=0.1) -> None:
        self.nodes = nodes #contains nodenames in format [node1, node2]
        self.arcs = arcs #contains arcs in the format [(node1, node2)]
        self.nodesValue = nodesValue #contains node capacity values in format
{node:cap}
        self.arcsInfo = arcsInfo #contains info about arcs in format {(node1,
node2): [length, weight, weighted_cost, lowerbound, upperbound]}
        self.paths = paths #contains the list of nodes connected in path for
reconstruction
        self.nodesCost = nodesCost #contains capture and storage cost for sources
and sinks in data in format {source:cap_cost, sink:storage_cost}
        self.duration = duration #duration of project
        self.target_cap = target_cap  #amount of C02 you want to be stored in
tCO₂/yr. note input will be given as MTCO₂/yr
        self.crf = crf


        self._initialize_sets()
        self._initialize_source_parameters()
        self._initialize_sink_parameters()
        self._initialize_arcs_parameters()
        self._initialize_pipeline_parameters()

        self.vars: Dict[str, gp.tupledict] = {}
        self.cons: Dict[str, gp.tupledict] = {}
```

```
        self.Big_M = 56.46 #max flow allowed in a pipeline tCO2/yr
        self.LTrend = 6.86 #upperbound flow for lower pipeline trend tCO2/yr

        self.costTrend = {"Slope": [0.1157192, 0.0783067],
                          "Intercept": [0.4316551, 0.770037]} #trends of pipeline
cost relating MTCO2/ to $M/yr

        self.c = len(self.costTrend["Slope"])




    def _initialize_sets(self) -> None:
        self.asset: Set = set() #all assets
        self.src: Set = set() #all source nodes
        self.sink: Set = set() #all storage nodes
        self.node: Set = set() #all transhipment nodes
        self.epipe: Set = set() #all existing pipelines
        self.a_a: Set = set() #all node to node connections
        self.two_way_arcs: Dict = set() #two way arcs, for (a, a') and (a', a) in
self.a_a, only take (a, a')


    def _initialize_source_parameters(self) -> None:
        self.source_annual_cap: Dict = {} #amount of CO2 that can be captured at
source annually (MtCO2/yr)
        self.capture_cost: Dict = {} #capture cost of CO2 at source in $/tCO2
        self.capture_fixed_cost: Dict = {} #fixed capture cost of CO2 at source
in $M
        self.capture_var_cost: Dict = {} #variable capture cost of CO2 at source
in $/tCO2

    def _initialize_sink_parameters(self) -> None:
        self.sink_cap: Dict = {} #total amount of CO2 that can be stored at a
sink in MTCO2
        self.storage_cost: Dict = {} #storage cost of CO2 at source in $/tCO2
        self.storage_fixed_cost: Dict = {} #fixed capture cost of CO2 at source
in $M
        self.storage_var_cost: Dict = {} #variable capture cost of CO2 at source
in $/tCO2

    def _initialize_arcs_parameters(self) -> None:
        self.max_arc_cap: Dict = {} #maximum amout of CO2 an arc/or pipeline can
transport annually (MtCO2/yr)
        self.min_arc_cap: Dict = {} #minimum amout of CO2 an arc/or pipeline can
transport annually (MtCO2/yr)
        self.arc_length: Dict = {} #length of arc/or pipeline in KM
        self.arc_weight: Dict = {} #weight of constructing arc. This corresponds
to the terrain
        self.arc_cost: Dict = {} #weighted cost of constructing arc (this is the
build cost)
```

```
    def _initialize_pipeline_parameters(self) -> None:
        self.pipe_nodes: Dict = {}

    def _generate_sets(self) -> None:
        self.asset = set(self.nodes)
        self.src = set([node for node in self.nodes if 'source' in node])
        self.sink = set([node for node in self.nodes if 'sink' in node])
        self.node = set([node for node in self.nodes if ((node not in self.src)
and (node not in self.sink))])
        self.epipe = set([node.split("_")[0] for node in self.nodes if ("_" in
node) and ('source' not in node) and ('sink' not in node)])
        self.a_a = set(self.arcs)

        #extract 2 way arcs
        seen = {}
        result = []

        for (a, b) in self.a_a:
            if (a, b) not in seen:
                seen[(a,b)] = True
                if (b, a) in seen:
                    result.append((b,a))

        self.two_way_arcs = set(result)

    def _generate_parameters(self) -> None:
        #source parameters
        self.source_annual_cap = {key:self.nodesValue[key] for key in self.src}
        self.capture_cost = {key:self.nodesCost[key][0] for key in self.src}
        self.capture_fixed_cost = {key:self.nodesCost[key][1] for key in
self.src}
        self.capture_var_cost = {key:self.nodesCost[key][2] for key in self.src}

        self.capture_v_cost = {key:self.capture_cost[key] if
(self.capture_var_cost[key] == 0)
                               and (self.capture_fixed_cost[key] == 0)
                               else self.capture_var_cost[key] for key in
self.src}

        #sink parameters
        self.sink_cap = {key:self.nodesValue[key] for key in self.sink}
        self.storage_cost = {key:self.nodesCost[key][0] for key in self.sink}
        self.storage_fixed_cost = {key:self.nodesCost[key][1] for key in
self.sink}
        self.storage_var_cost = {key:self.nodesCost[key][2] for key in self.sink}

        self.storage_v_cost = {key:self.storage_cost[key] if
(self.storage_var_cost[key] == 0)
                               and (self.storage_fixed_cost[key] == 0)
                               else self.storage_var_cost[key] for key in
self.sink}
```

```
        #arc parameters
        self.MaxCap = sum(self.source_annual_cap.values()) #maximum possible flow
        self.MidCap = ((self.costTrend["Intercept"][1] -
self.costTrend["Intercept"][0]) / (self.costTrend["Slope"][0] -
self.costTrend["Slope"][1]))


        # self.max_arc_cap = {key:self.arcsInfo[key][4] for key in self.a_a}
        self.max_arc_cap = {(akey[0], akey[1], ckey):self.arcsInfo[akey][4] if
self.arcsInfo[akey][4] < self.MidCap else self.MidCap if ckey == 0 else
self.MaxCap
                            for akey in self.a_a for ckey in range(self.c)}

        # self.min_arc_cap = {key:self.arcsInfo[key][3] for key in self.a_a}
        self.min_arc_cap = {(akey[0], akey[1], ckey):self.arcsInfo[akey][3] if
self.arcsInfo[akey][3] > 0 else 0
                            for akey in self.a_a for ckey in range(self.c)}
        self.arc_length = {key:self.arcsInfo[key][0] for key in self.a_a}
        self.arc_weight = {key:self.arcsInfo[key][1] for key in self.a_a}
        self.arc_cost = {key:self.arcsInfo[key][2] for key in self.a_a}


        #pipeline parameters
        self.pipe_nodes = {key:[pipenode for pipenode in self.node if key in
pipenode] for key in self.epipe}

    def _validation_checks(self) -> None:
        #if target cap greater than total source cap, then set target cap to
source cap
        total_source_cap = sum(self.source_annual_cap.values()) #MTCO₂/yr
        total_sink_cap = -sum(self.sink_cap.values()) / self.duration #MTCO₂/yr
        total_max_arc_flow = sum(self.max_arc_cap.values()) #MTCO₂/yr

        # print(total_source_cap, total_sink_cap, total_max_arc_flow)
        LOGGER.info(f'Target capacity (MTCO₂/yr): {self.target_cap}')
        LOGGER.info(f'Total source capacity (MTCO₂/yr): {total_source_cap}')
        LOGGER.info(f'Total sink capacity (MTCO₂/yr): {total_sink_cap}')
        LOGGER.info(f'Total pipe capacity (MTCO₂/yr): {total_max_arc_flow}')

        limiting_flow = min(total_source_cap, total_sink_cap, total_max_arc_flow)

        LOGGER.info(f'Limiting Flow (MTCO₂/yr): {limiting_flow}')

        if self.target_cap > limiting_flow:
            LOGGER.warning('Target cap greater than limiting flow, resetting
target to limiting flow')
            self.target_cap = limiting_flow


    def create_sets_and_parameters(self):
        self._generate_sets()
```

```
        self._generate_parameters()
        self._validation_checks()


    def create_variables(self) -> None:
        #flow from node 1 to node 2 in network (tCO₂/yr)
        index = ((node1, node2, c) for (node1, node2) in self.a_a for c in
range(self.c))
        self.vars['arc_flow'] = self.model.addVars(index, name='arc_flow', lb=0,
vtype=GRB.CONTINUOUS)

        #amount of CO₂ captured at source (tCO₂/yr)
        index = (src for src in self.src)
        self.vars['CO₂_captured'] = self.model.addVars(index,
name='CO₂_captured', lb=0, vtype=GRB.CONTINUOUS)

        #amount of CO₂ stored at sink (tCO₂/yr)
        index = (sink for sink in self.sink)
        self.vars['CO₂_injected'] = self.model.addVars(index,
name='CO₂_injected', lb=0, vtype=GRB.CONTINUOUS)

        #indicator for if pipeline arc connecting node 1 to 2 is built
        index = ((node1, node2, c) for (node1, node2) in self.a_a for c in
range(self.c))
        self.vars['arc_built'] = self.model.addVars(index, name='arc_built',
vtype=GRB.BINARY)

        #indicator is source is opened
        index = (src for src in self.src)
        self.vars['src_opened'] = self.model.addVars(index, name='src_opened',
vtype=GRB.BINARY)

        #indicator is sink is opened
        index = (sink for sink in self.sink)
        self.vars['sink_opened'] = self.model.addVars(index, name='sink_opened',
vtype=GRB.BINARY)



    def _initialize_gurobi(self) -> None:
        self.env = gp.Env(empty=True)
        self.env.start()
        self.model = gp.Model("CO₂_network_optimization", env=self.env)


    def _arc_upper_lower_bound_cons(self) -> None:
        cons_name = 'arc_lower_bound'
        constr = ((self.min_arc_cap[node1, node2, c]) *
self.vars['arc_built'][node1, node2, c] #conversion min cap from MTCO₂/yr to
tCO₂/yr
                    <= self.vars['arc_flow'][node1, node2, c]
                    for (node1, node2) in self.a_a
```

```
                    for c in range(self.c))
        self.cons[cons_name] = self.model.addConstrs(constr, name=cons_name)

        cons_name = 'arc_upper_bound'
        constr = ((self.max_arc_cap[node1, node2, c]) *
self.vars['arc_built'][node1, node2, c] #conversion max cap from MTCO₂/yr to
tCO₂/yr
                    >= self.vars['arc_flow'][node1, node2, c]
                    for (node1, node2) in self.a_a
                    for c in range(self.c))
        self.cons[cons_name] = self.model.addConstrs(constr, name=cons_name)


    def _single_direction_arc_flow_cons(self) -> None:
        cons_name = 'arc_single_dir_flow'
        constr = (sum(self.vars['arc_built'][node1, node2, c] for c in
range(self.c)) <= 1
                  for (node1, node2) in self.a_a)
        self.cons[cons_name] =  self.model.addConstrs(constr, name=cons_name)


    def _node_balance_cons(self) -> None:
        asset_to_node = {n:[a for a in self.asset
                            if (a,n) in self.a_a]
                         for n in self.node}
        node_to_asset = {n:[a for a in self.asset
                            if (n,a) in self.a_a]
                         for n in self.node}

        cons_name = 'node_balance'
        constr = (sum(self.vars['arc_flow'][a,n,c1] for a in asset_to_node[n] for
c1 in range(self.c))
                    == sum(self.vars['arc_flow'][n,a,c2] for a in
node_to_asset[n] for c2 in range(self.c))
                    for n in self.node)
        self.cons[cons_name] = self.model.addConstrs(constr, name=cons_name)


    def _demand_balance_cons(self) -> None:
        asset_to_demand = {d:[a for a in self.asset
                            if (a,d) in self.a_a]
                         for d in self.sink}
        demand_to_asset = {d:[a for a in self.asset
                            if (d,a) in self.a_a]
                         for d in self.sink}

        cons_name = 'demand_balance'
        constr = (sum(self.vars['arc_flow'][a,d,c1] for a in asset_to_demand[d]
for c1 in range(self.c))*self.duration  #convert tCO₂/yr to MTCO₂
                    - sum(self.vars['arc_flow'][d,a,c2] for a in
demand_to_asset[d] for c2 in range(self.c))*self.duration
                    == self.vars['CO₂_injected'][d] #MTCO₂
```

```
                    for d in self.sink)
        self.cons[cons_name] = self.model.addConstrs(constr, name=cons_name)


    def _supply_balance_cons(self) -> None:
        asset_to_supply = {s:[a for a in self.asset
                              if (a,s) in self.a_a]
                           for s in self.src}
        supply_to_asset = {s:[a for a in self.asset
                              if (s,a) in self.a_a]
                           for s in self.src}

        cons_name = 'supply_balance'
        constr = (sum(self.vars['arc_flow'][a,s,c1] for a in asset_to_supply[s]
for c1 in range(self.c))  #convert tCO₂/yr to MTCO₂/yr
                    - sum(self.vars['arc_flow'][s,a,c2] for a in
supply_to_asset[s] for c2 in range(self.c))
                    == -self.vars['CO₂_captured'][s] #MTCO₂/yr
                    for s in self.src)
        self.cons[cons_name] = self.model.addConstrs(constr, name=cons_name)


    def _capture_limit_cons(self) -> None:
        cons_name = 'capture_limit'
        constr = (self.vars['CO₂_captured'][s] #MTCO₂/yr
                    <= self.source_annual_cap[s] * self.vars['src_opened'][s]
                    for s in self.src)
        self.cons[cons_name] = self.model.addConstrs(constr, name=cons_name)


    def _storage_limit_cons(self) -> None:
        cons_name = 'storage_limit'
        constr = (self.vars['CO₂_injected'][d]
                    <= -self.sink_cap[d] * self.vars['sink_opened'][d] #1e6
converts MTCO₂ to tCO₂
                    for d in self.sink)
        self.cons[cons_name] = self.model.addConstrs(constr, name=cons_name)


    def _capture_target_cons(self) -> None:
        cons_name = 'CO₂_capture_target'
        constr = (sum(self.vars['CO₂_captured'][s] for s in self.src)
                    >= self.target_cap)
        self.cons[cons_name] = self.model.addConstr(constr, name=cons_name)







    def create_constraints(self) -> None:
        self._arc_upper_lower_bound_cons()
```

```
    msg = ("'Arc Bounds' constraint: Time elapsed: %.2f seconds"
           % (time.time() - START_TIME))
    print(msg)
    LOGGER.info(msg)
    self._single_direction_arc_flow_cons()
    msg = ("'Single Direction' constraint: Time elapsed: %.2f seconds"
           % (time.time() - START_TIME))
    print(msg)
    LOGGER.info(msg)
    self._node_balance_cons()
    msg = ("'Supply Balance' constraint: Time elapsed: %.2f seconds"
           % (time.time() - START_TIME))
    print(msg)
    LOGGER.info(msg)
    self._demand_balance_cons()
    msg = ("'Demand Balance' constraint: Time elapsed: %.2f seconds"
           % (time.time() - START_TIME))
    print(msg)
    LOGGER.info(msg)
    self._supply_balance_cons()
    msg = ("'Supply Balance' constraint: Time elapsed: %.2f seconds"
           % (time.time() - START_TIME))
    print(msg)
    LOGGER.info(msg)
    self._capture_limit_cons()
    msg = ("'Capture Limit' constraint: Time elapsed: %.2f seconds"
           % (time.time() - START_TIME))
    print(msg)
    LOGGER.info(msg)
    self._storage_limit_cons()
    msg = ("'Storage Limit' constraint: Time elapsed: %.2f seconds"
           % (time.time() - START_TIME))
    print(msg)
    LOGGER.info(msg)
    self._capture_target_cons()
    msg = ("'Capture Target' constraint: Time elapsed: %.2f seconds"
           % (time.time() - START_TIME))
    print(msg)
    LOGGER.info(msg)



def build_model(self) -> None:
    self._initialize_gurobi()
    print('\nInitialized Gurobi model instance\n')
    LOGGER.info('\nInitialized Gurobi model instance')
    LOGGER.info('Creating sets and parameters...')
    print('Creating sets and parameters...')
    self.create_sets_and_parameters()
    print('Sets and parameters are generated')
    print("Time elapsed: %.2f seconds" % (time.time() - START_TIME))
```

```
        LOGGER.info('Sets and parameters are generated')
        LOGGER.info("Time elapsed: %.2f seconds" % (time.time() - START_TIME))
        LOGGER.info('Setting variables...')
        print('\nSetting variables...')
        self.create_variables()
        print('Variables are defined')
        print("Time elapsed: %.2f seconds" % (time.time() - START_TIME))
        LOGGER.info('Variables are defined')
        LOGGER.info("Time elapsed: %.2f seconds" % (time.time() - START_TIME))
        LOGGER.info('Imposing constraints...')
        print('\nImposing constraints...')
        self.create_constraints()
        print('Constraints are enforced\n')
        LOGGER.info('Constraints are enforced')
        print('Solving model...\n')


    def create_objective(self) -> None:
        #capture cost + transport flow cost + arc build cost + storage cost
        capture_cost = sum((self.capture_fixed_cost[s] *
self.vars['src_opened'][s]) + # $M * {0,1} = $M
                           (self.capture_v_cost[s] *
self.vars['CO₂_captured'][s] * self.duration) for s in self.src) # $/tCO₂ *
MTCO₂/yr * yr = $M

        storage_cost = sum((self.storage_fixed_cost[d] *
self.vars['sink_opened'][d]) + # $M * {0,1} = $M
                           (self.storage_v_cost[d] *
self.vars['CO₂_injected'][d]) for d in self.sink) # $/tCO₂ * MTCO₂ = $M


        transport_flow_cost = sum((self.costTrend["Slope"][c] *
self.vars['arc_flow'][node1, node2, c])
                                  *  self.arc_cost[node1, node2] * self.crf *
self.duration
                                       for (node1, node2) in self.a_a
                                       for c in range(self.c)) # $M * {0, 1} =
$M



        pipeline_build_cost = sum((self.costTrend["Intercept"][c] *
self.vars['arc_built'][node1, node2, c])
                                  *  self.arc_cost[node1, node2] * self.crf *
self.duration
                                       for (node1, node2) in self.a_a
                                       for c in range(self.c)) # $M * {0, 1} =
$M

        obj =  capture_cost + storage_cost + transport_flow_cost +
pipeline_build_cost
```

```
        self.model.setObjective(obj, GRB.MINIMIZE)
        self.model.update()

    def solve_model(self) -> None:
        LOGGER.info('Evauating "minimum cost" objective function')
        self.create_objective()
        LOGGER.info('Objective function "mimumum cost" evaluated')
        self.use_pulp = False

        #set numrerical focus to 2
        # self.model.setParam('NumericFocus', 2)
        #output lp and mps files
        self.model.write(LP_FILE_PATH)
        self.model.write(MPS_FILE_PATH)

        if (self.model.NumVars <= 2000) and (self.model.NumConstrs <= 2000):
            #solve model
            self.model.optimize()
            LOGGER.info(f'Model Status: {self.model.status}')
            if self.model.status == GRB.INFEASIBLE:
                self.model.computeIIS()
                self.model.write(ILP_FILE_PATH)
            elif self.model.status == GRB.INF_OR_UNBD:
                self.model.setParam('DualReductions', 0)
                self.model.optimize()
                if self.model.status == GRB.INFEASIBLE:
                    self.model.computeIIS()
                    self.model.write(ILP_FILE_PATH)
            else:
                self.objective = self.model.ObjVal

                #write solution
                self.model.write(SOL_FILE_PATH)
                self.extract_results()
            LOGGER.info("Time elapsed: %.2f seconds" % (time.time() -
START_TIME))
        else:
            LOGGER.info("Model is too large for Gurobipy free licence, switching
to CPLEX")
            self.use_pulp=True
            self.pulp_var, self.pulp_model = LpProblem.fromMPS(MPS_FILE_PATH)
            self.pulp_solver = pl.CPLEX_CMD(options=['mipdisplay=0'])
            self.pulp_model.solve(self.pulp_solver)
            if self.pulp_model.status == 1:
                #write soln
                self.extract_pulp_variables()
                self.extract_results()
            LOGGER.info("Time elapsed: %.2f seconds" % (time.time() -
START_TIME))
```

```
    def extract_pulp_variables(self) -> None:
        prob1 = self.pulp_model
        arc_flow_keys = {}
        CO₂_captured_keys = {}
        CO₂_injected_keys = {}
        arc_built_keys = {}
        src_opened_keys = {}
        sink_opened_keys = {}

        for v in prob1.variables():
            if "arc_flow" in v.name:
                key1 = v.name.split(",")[0][9:]
                key2 = v.name.split(",")[1]
                key3 = int(v.name.split(",")[2].split("_")[0])
                arc_flow_keys[(key1, key2, key3)] = v.varValue
            if "CO₂_captured" in v.name:
                key = v.name.split("_")[2] + "_" + v.name.split("_")[3]
                CO₂_captured_keys[key] = v.varValue
            if "CO₂_injected" in v.name:
                key = v.name.split("_")[2] + "_" + v.name.split("_")[3]
                CO₂_injected_keys[key] = v.varValue
            if "arc_built" in v.name:
                key1 = v.name.split(",")[0][10:]
                key2 = v.name.split(",")[1]
                key3 = int(v.name.split(",")[2].split("_")[0])
                arc_built_keys[(key1, key2, key3)] = v.varValue
            if "src_opened" in v.name:
                key = v.name.split("_")[2] + "_" + v.name.split("_")[3]
                src_opened_keys[key] = v.varValue
            if "sink_opened" in v.name:
                key = v.name.split("_")[2] + "_" + v.name.split("_")[3]
                sink_opened_keys[key] = v.varValue

        # Write the solution to a .sol file
        with open(SOL_FILE_PATH, "w") as f:
            f.write("# Solution for model CO₂_network_optimization \n")
            f.write(f"# Objective value = {value(prob1.objective)} \n")
            for key in arc_flow_keys.keys():
                f.write(f"arc_flow[{key[0]},{key[1]},{key[2]}]
{arc_flow_keys[key]} \n")
            for key in CO₂_captured_keys.keys():
                f.write(f"CO₂_captured[{key}] {CO₂_captured_keys[key]} \n")
            for key in CO₂_injected_keys.keys():
                f.write(f"CO₂_injected[{key}] {CO₂_injected_keys[key]} \n")
            for key in arc_built_keys.keys():
                f.write(f"arc_built[{key[0]},{key[1]},{key[2]}]
{int(arc_built_keys[key])} \n")
            for key in src_opened_keys.keys():
                f.write(f"src_opened[{key}] {int(src_opened_keys[key])} \n")
            for key in sink_opened_keys.keys():
                f.write(f"sink_opened[{key}] {int(sink_opened_keys[key])} \n")
```

```
        self.arc_flow_keys = arc_flow_keys
        self.CO2_captured_keys = CO2_captured_keys
        self.CO2_injected_keys = CO2_injected_keys
        self.arc_built_keys = arc_built_keys
        self.src_opened_keys = src_opened_keys
        self.sink_opened_keys = sink_opened_keys




    def extract_soln_arcs(self) -> None:
        self.soln_arcs = {}
        for arc in self.vars['arc_flow']:
            if self.vars['arc_flow'][arc].X > 0:
                self.soln_arcs[arc] = self.vars['arc_flow'][arc].X

        self.soln_arcs_a = {(arc[0], arc[1]):self.soln_arcs[arc] for arc in
self.soln_arcs.keys()}

    def extract_activated_source(self) -> None:
        self.soln_sources = {}
        for src in self.vars['CO2_captured']:
            if self.vars['CO2_captured'][src].X > 0:
                self.soln_sources[src] = self.vars['CO2_captured'][src].X

    def extract_activated_sinks(self) -> None:
        self.soln_sinks = {}
        for sink in self.vars['CO2_injected']:
            if self.vars['CO2_injected'][sink].X > 0:
                self.soln_sinks[sink] = self.vars['CO2_injected'][sink].X

    def extract_costs(self) -> None:
        self.soln_cap_costs = {} #$M
        self.soln_storage_costs = {} #$M
        self.soln_transport_costs = {}

        for src in self.soln_sources.keys():
            c_cost = (self.capture_fixed_cost[src] + (self.capture_v_cost[src] *
self.soln_sources[src] * self.duration))
            self.soln_cap_costs[src] = c_cost

        for sink in self.soln_sinks.keys():
            s_cost = self.storage_fixed_cost[sink] + (self.storage_v_cost[sink] *
self.soln_sinks[sink])
            self.soln_storage_costs[sink] = s_cost

        for arc in self.soln_arcs.keys():
            print("arc: ", arc)
            print("slope: ", self.costTrend["Slope"][arc[2]])
            print("intercept: ", self.costTrend["Intercept"][arc[2]])
            print("flow: ", self.vars['arc_flow'][arc].x)
```

```
            print("built: ", self.vars['arc_built'][arc].x)
            print("weight: ", self.arc_cost[(arc[0], arc[1])])
            print("crf: ", self.crf)
            print("duration: ", self.duration)


            tf_cost = (self.costTrend["Slope"][arc[2]] *
self.vars['arc_flow'][arc].x) *  self.arc_cost[(arc[0], arc[1])] * self.crf *
self.duration
            tb_cost = (self.costTrend["Intercept"][arc[2]] *
self.vars['arc_built'][arc].x) *  self.arc_cost[(arc[0], arc[1])] * self.crf *
self.duration

            t_cost = tf_cost + tb_cost

            print("transfer: ", tf_cost)
            print("build: ", tb_cost)
            print("total: ", t_cost)
            print("")

            self.soln_transport_costs[arc] = t_cost

            self.soln_transport_costs_a = {(arc[0],
arc[1]):self.soln_transport_costs[arc] for arc in
self.soln_transport_costs.keys()}


    def extract_soln_arcs_p(self) -> None:
        self.soln_arcs = {}
        for arc in self.arc_flow_keys.keys():
            if self.arc_flow_keys[arc] > 0:
                self.soln_arcs[arc] = self.arc_flow_keys[arc]

        self.soln_arcs_a = {(arc[0], arc[1]):self.soln_arcs[arc] for arc in
self.soln_arcs.keys()}

    def extract_activated_source_p(self) -> None:
        self.soln_sources = {}
        for src in self.CO2_captured_keys.keys():
            if self.CO2_captured_keys[src] > 0:
                self.soln_sources[src] = self.CO2_captured_keys[src]

    def extract_activated_sinks_p(self) -> None:
        self.soln_sinks = {}
        for sink in self.CO2_injected_keys.keys():
            if self.CO2_injected_keys[sink] > 0:
                self.soln_sinks[sink] = self.CO2_injected_keys[sink]


    def extract_costs_p(self) -> None:
        self.soln_cap_costs = {} #$M
        self.soln_storage_costs = {} #$M
```

```
        self.soln_transport_costs = {}

        for src in self.soln_sources.keys():
            c_cost = (self.capture_fixed_cost[src] + (self.capture_v_cost[src] *
self.soln_sources[src] * self.duration))
            self.soln_cap_costs[src] = c_cost

        for sink in self.soln_sinks.keys():
            s_cost = self.storage_fixed_cost[sink] + (self.storage_v_cost[sink] *
self.soln_sinks[sink])
            self.soln_storage_costs[sink] = s_cost

        for arc in self.soln_arcs.keys():
            print("arc: ", arc)
            print("slope: ", self.costTrend["Slope"][arc[2]])
            print("intercept: ", self.costTrend["Intercept"][arc[2]])
            print("flow: ", self.arc_flow_keys[arc])
            print("built: ", self.arc_built_keys[arc])
            print("weight: ", self.arc_cost[(arc[0], arc[1])])
            print("crf: ", self.crf)
            print("duration: ", self.duration)
            tf_cost = (self.costTrend["Slope"][arc[2]] * self.arc_flow_keys[arc])
*  self.arc_cost[(arc[0], arc[1])] * self.crf * self.duration
            tb_cost = (self.costTrend["Intercept"][arc[2]] *
self.arc_built_keys[arc]) *  self.arc_cost[(arc[0], arc[1])] * self.crf *
self.duration

            t_cost = tf_cost + tb_cost

            print("transfer: ", tf_cost)
            print("build: ", tb_cost)
            print("total: ", t_cost)
            print("")

            self.soln_transport_costs[arc] = t_cost

            self.soln_transport_costs_a = {(arc[0],
arc[1]):self.soln_transport_costs[arc] for arc in
self.soln_transport_costs.keys()}




    def extract_results(self) -> None:
        if self.use_pulp:
            self.extract_soln_arcs_p()
            self.extract_activated_source_p()
            self.extract_activated_sinks_p()
            self.extract_costs_p()
        else:
            self.extract_soln_arcs()
            self.extract_activated_source()
```

```
            self.extract_activated_sinks()
            self.extract_costs()

    def get_soln_arcs(self):
        return self.soln_arcs_a

    def get_soln_sources(self):
        return self.soln_sources

    def get_soln_sinks(self):
        return self.soln_sinks

    def get_soln_cap_costs(self):
        return self.soln_cap_costs

    def get_soln_storage_costs(self):
        return self.soln_storage_costs

    def get_soln_transport_costs(self):
        return self.soln_transport_costs_a

    def get_all_soln_results(self):
        return self.soln_arcs_a, self.soln_sources, self.soln_sinks,
self.soln_cap_costs, self.soln_storage_costs, self.soln_transport_costs_a
```