MSc Computer Science

University of Birmingham – School of Computer Science

2025



**Causal Compass: A Beginner-Friendly GUI for Causal Discovery**

Jonah King

2755137

Supervised by Dr. Montero-Hernandez

Contents

[Abstract 3](#_Toc204788838)

[Acknowledgements 3](#_Toc204788839)

[Chapter 1: Introduction 4](#_Toc204788840)

[Chapter 2: Research & Background 6](#_Toc204788841)

[Chapter 3: Project Specification 9](#_Toc204788843)

[Chapter 4: Design 16](#_Toc204788844)

[Chapter 5: Project Management 18](#_Toc204788845)

[Chapter 6: Implementation 19](#_Toc204788846)

[Chapter 7: Testing 23](#_Toc204788847)

[Chapter 8: Evaluation 25](#_Toc204788848)

[Chapter 9: Conclusion 37](#_Toc204788849)

[References 38](#_Toc204788850)

[Appendix 41](#_Toc204788851)

**Declaration:** I confirm that this project is my own work. Code development and debugging was assisted by ChatGPT o3. The report’s main body and figures are mine with assistance from ChatGPT o4-mini for reorganising the report’s structure and editing sections for conciseness. Everything submitted has been reviewed by me to reflect my own work.

# ***Abstract***

This project develops a beginner-oriented, offline desktop GUI which is accessible to students and novice researchers allowing them to run and interpret standard causal discovery algorithms. The application wraps PC, GFCI, and FGES via py-tetrad, renders causal graphs with consistent edge-mark semantics, and has a workflow focused on explainability. The design follows evidence-based usability principles (progressive disclosure, clear labelling, and visible status) and runs entirely offline.

Implementation was validated end-to-end using LUCAS, ASIA, and Sachs datasets. Unit and integration testing confirmed robustness and a task-based study with 7 novices reported high usability. Performance testing consistently passed well within targeted measurements, validating the UI’s responsiveness. All must have functional requirements were successfully implemented and most non-functional requirements passed.

Limitations include narrow algorithm coverage and small-scale user testing. Future work will aim to broaden methods, add UI refinement features, improve setup guidance, and perform cross platform evaluation. Overall, the tool provides a novice friendly on-ramp to causal discovery for teaching and early-stage causal analysis.

# ***Acknowledgements***

I would like to thank my supervisor, Dr Samuel Antonio Montero-Hernandez, for providing continuous support throughout the entirety of my Master’s project. His advice and interest in my work exceeded my expectations and has helped me get a lot more out of the project. I am also thankful for the support from family and friends which has not gone unrecognised.

## Glossary

Concise definitions for key terms used throughout this report.

* **Observational vs interventional data:** *Observational data* are collected passively (no controlled experiments). Whereas *interventional data* come from controlling a variable (e.g. Do(X=x). Observational data is focused on in this project.
* **Causal graph / causal model:** A graph where each node is a variable and each edge between nodes represents a possible causal relationship under the model’s assumptions.
* **DAG (Directed Acyclic Graph):** A causal graph with arrows and no cyclical relationships (no arrow paths circle back to themselves). Encodes which variable causes another.
* **CPDAG (Completed Partially Directed Acyclic Graph):** A summary of all DAGs which all fit the data equally as well (this is a Markov equivalence class). Contains directed and undirected edges where orientation can’t be deduced purely from data/assumptions.
* **PAG (Partial Ancestral Graph):** like CPDAG except hidden confounders are allowed. Special endpoint markers with circles and double headed arrows show uncertainty and latent causes.
* **Edge marks shown in the GUI:**

A --> B: A is very likely a cause of B (ruling out B causing A).

A --- B: Undirected adjacency (unresolved orientation in a CPDAG).

A o-> B: Certain arrowhead at B but uncertain at A (either A --> B or A <-> B).

A o-o B: Both ends are uncertain. This is an adjacency that can’t be oriented.

A <-> B: Association is likely explained by an unmeasured confounder.

* **d-separation:** a rule which decides when variables should be independent given other variables. Used by constraint-based algorithms when deciding to remove, add or orient edges.
* **Conditional independence test:** A statistical test which constraint-based methods use. Fisher-Z for continuous, roughly Gaussian data and χ² / G² for discrete data.
* **Score / scoring function:** A score which trades fit vs complexity. These methods search for the graph which gives the best score, e.g. Bayesian information criterion (BIC)
* **Constraint-based vs score-based vs hybrid:**

*Constraint-based* (e.g. PC) use independence tests + α.

*Score-based* (e.g. FGES) optimise an appropriate score (e.g. BIC) with the given data.

*Hybrid* (e.g. GFCI) combines constraint and score-based methods and outputs PAGs.

* **PC algorithm:** constraint-based search algorithm which outputs a CPDAG. Key parameters include α (significance value) and depth (maximum size of conditioning sets)
* **FGES algorithm:** score-based greedy search algorithm which outputs a CPDAG computed using the best scoring equivalence class. The key parameter is penalty discount that controls sparsity (higher sparsity = fewer edges, lower sparsity = more edges) of the CPDAG.
* **GFCI algorithm:** hybrid search algorithm that allows representation of confounding variables. This outputs a PAG and uses independence tests (like PC) alongside scoring (like FGES).
* **α (alpha, significance value):** The threshold for independence tests. A smaller α = fewer edges. A larger α = more edges
* **depth (maximum conditioning set size):** this limits how many variables an independence test conditions on. Lower depth = faster processing time and stabilises testing. -1 is used for no limit.
* **Penalty discount (used by FGES):** determines the complexity penalty in BIC-like scores. Higher values = sparser graphs = fewer edges.
* **Separating set (sepset):** S is a set of variables that means A and B are independent when conditioned on (A ⟂ B | S). The “Why not connected?” tab in the GUI calculates a sepset for chosen variables and displays one if found.
* **Latent confounder / causal sufficiency:** a latent confounder is an unmeasured common cause of two variables. Causal sufficiency assumes there are no hidden causes, meaning no latent confounders exist outside of the given data. Many real-world datasets violate this assumption which is why GFCI is useful and outputted PAGs matter.
* **Markov assumption (Causal Markov Condition):** once you know a node’s direct causes and its parents in a DAG, non-descendants add no extra information about this variable. This allows independencies to be read from the graph. If A --> B --> C, A and C are generally dependent, but the Markov assumption states A tells you nothing about C
* **Faithfulness assumption:** all conditional independencies in the data come from the graph’s structure. It states there are no coincidental rulings out of any causal effects in the DAG.
* **Collider & back-door path:** A collider is an A --> C <-- B relationship. This creates a spurious association between A and B. A back-door path from A to B is any path conditioning on colliders opens paths. Blocking back-door paths with conditioning removes confounding variables.

These are terms consistently used in the GUI and in this report which readers should understand to follow the design choices, results, and limitations of this project.

# ***Chapter 1: Introduction***

Causal discovery is used by researchers to reason about potential cause and effect structures using observation data. However, current tools frequently require expert technical skills in statistics and software. This project addresses this gap by developing a beginner-oriented desktop GUI that provides important guidance. New users are walked through loading a CSV, running standard searches (PC, GFCI, FGES via py-tetrad), reading and interpreting graph semantics, and exporting reproducible outputs. The application can run entirely offline on a typical student laptop making it accessible for novice users in education. This report describes the design, implementation, and evaluation of this prototype.

## 1.1 Motivation & Context

Students and non-specialist researchers are increasingly working with observational datasets across many disciplines. However, three barriers are faced by beginners when trying causal discovery. i) parameter anxiety (uncertainty over α, depth, and test/score type choices), ii) opaque outputs (CPDAG/PAGs can be difficult to interpret), and iii) workflow friction (convoluted steps for running algorithms, comparing results, and exporting figures). Well established tools (e.g. TETRAD) implement a wide range of algorithms and have expert control, causing a steep learning curve for novices. A guided and focused GUI can lower the barrier to entry by pairing sensible default settings and tutorials alongside prioritising readable graphs, one-click exports, supporting teaching and early-stage research, and reproducible coursework.

## 1.2 Aim

The aim of this project is to create an offline, beginner-focused GUI for causal discovery to enable a first-time user to: import a CSV, choose and configure PC/GFCI/FGES with well explained parameters, understand CPDAG/PAGs edge-marks, compare algorithms, and export graphs and reports for reproducibility. The prototype is evaluated against defined requirements via unit and integration testing, a small beginner user study, performance timings, and case studies on LUCAS, ASIA, and Sachs datasets.

# ***Chapter 2: Research & Background***

## 2.1 Foundations of Causality

Causality goes beyond simple association in explaining real world phenomena. Two variables can be correlated without one causing the other. This can be explained by confounding variables, selection effects or possible coincidences. Structural Causal Models (SCMs) formalise cause and effect relationships as structural equations and can represent these with Directed Acyclic Graphs (DAGs) (Maathuis, 2009). These encode which variables directly cause other variables. With these rules, missing arrows in the graph imply conditional independencies between variables via the concept of d-separation (Spirtes, 2001). These independencies are what causal discovery algorithms search for in observational data

There are two core assumptions which link data to the learned graph. The *Markov assumption* states that each variable is independent of its non-descendants given its parents in the true DAG (Pearl, 2009). This implies that d-separations map to conditional independencies in the population distribution (Spirtes, 2001). The *faithfulness assumption* posits that all statistical independencies observed in the data are direct consequences of the underlying causal structure. They are not a coincidence or result of cancellation effects. This means conditional independencies implied by the causal graph match the observed conditional independencies. Together these assumptions allow inference of causal structure from observational data using conditional independence tests. In practice, this yields an equivalence class of DAGs (a CPDAG), for which intervention effects can still be estimated via intervention calculus (IDA) (Maathuis, 2009).

Interventions are represented by *do-operations.* Interventional distributions differ from observational data and resolve ambiguities that observational data cannot on its own. In practice, causal discovery, purely from observational data, involves recovering an equivalence class of DAGs which share the same independencies. When allowing for latent confounding variables, the target output becomes a *Partial Ancestral Graph (PAG).* The edge marks in PAGs allow for uncertainty and unmeasured hidden causes.

These ideas motivate the algorithms my GUI wraps. PC is a constraint-based method which starts from a complete undirected graph (Spirtes, 2001). Edges are removed and oriented using conditional-independence tests (Fisher-Z for Gaussian data, χ² / G² for discrete), The output is a CPDAG which represents a DAG equivalence class under Markov and faithfulness assumptions. Greedy Fast Causal Inference (GFCI) is a hybrid algorithm that combines both constraint-based FCI with score-based search (Ogarrio, 2016). This aims to improve robustness with latent confounding variables and outputs a PAG. FGES (Fast Greedy Equivalence Search) is a score-based method. It searches equivalence classes choosing the one which most optimises a score (e.g. Bayesian Information Criterion (BIC)) and outputs a CPDAG (Chickering, 2002)(Ramsey, 2017). My GUI interface conveys these differences by showing edge endpoint mark semantics and by saving the test/score choices in captions and exported figures.

**Why this matters for the GUI:** the independence-based logic (d-separation), the equivalence classes, and the role of assumptions (Markov, faithfulness, possible latent confounding variables) explain why there may be disagreements across algorithms on edge direction or “uncertain” relationships. Designing a GUI for novice users therefore requires clear labelling of algorithm types (score vs constraint-based), display of used parameters/tests/scores, and plain English explanations of why an edge is present/absent. This is exactly what the edge explanation, “Why not connected?” and algorithm comparison tabs provide (Chapter 7).

## 2.2 Causal Discovery Algorithms, Data Types, Tests & Scores

This project implements a small set of causal discovery algorithms deliberately chosen for novice users to learn the core ideas while avoiding the risk of overwhelming the user. The GUI involves one constraint-based method (PC), one score-based method (FGES), and one hybrid method (GFCI). These three algorithms cover common assumptions in causal modelling and output consistently rendered and explainable graphs.

**Algorithms covered:**

***PC (constraint-based; CPDAG output):***searches for a graph in two phases: 1) *Skeleton phase* – remove edges that contradict conditional independence (CI) tests. This progressively conditions on larger sets up to the depth controlled by the user, and 2) ***Orientation phase*** – collider detection and Meek’s rules are applied to orient edges which the data allows. *Output*: CPDAG (completed partially directed acyclic graph). This represents the Markov equivalence class of DAGs. The GUI renders directed arrows for fixed orientation edges and undirected endpoints when no direction is determined. *Parameters exposed by the GUI***:** significance level α, depth, and data type (appropriate test).

***GFCI (hybrid; PAG output; latent confounding allowed):*** GFCI begins with a score-based search to find a graph structure**. Edges are validated and pruned via CI tests. These are then oriented. Due to GFCI being able to reason about possible unmeasured confounding variables, edges can be partially oriented. *Output:*** PAG (partial ancestral graph) which captures uncertainty due to latent confounding variables with uncertain endpoints. ***Parameters exposed*: α**, **depth**, and **data type** (appropriate test/score).

***FGES (score-based; CPDAG output):*** FGES optimises a global score with a greedy equivalence search involving both a forward (adding edges) and backward (removing edges) pass. Edges are then oriented to match the equivalence class. ***Output:*** CPDAG. ***Parameters exposed:*** **penalty discount *c* for BIC-style score and data type (appropriate score). FGES does not use** α or depth.

**Data types and statistical tests**

The GUI supports two broad data types. The choice determines both scores (for FGES) and CI tests (for PC/GFCI).

***Continuous data:***

* **CI test: Fisher-Z test on partial correlation. Residuals are computed for a candidate separating set *S*. The Pearson *r* is taken of residuals and transformed with Fisher-Z. A two-sided *p-value* is then compared to** α (Kalisch, 2005).
* Score: SEM-BIC is a linear Gaussian structural equation modelling with BIC penalty (Schwarz, 1978). The penalty discount *c* scales complexity. A large c = sparser graphs (fewer edges).

***Discrete data:***

* CI test: conditional χ² / G² test aggregated over a strata of S. Independence is accepted when p≥α.
* **Score: BDeu** (or discrete BIC variants). Also controlled by the **penalty discount** in FGES.

**Auto detect in the GUI**

When users select “auto”, columns are classified via simple transparent heuristics. Integer columns with small cardinality are treated as discrete data. Otherwise, data is treated as continuous. This sets both the CI test (PC/GFCI) and the score family (FGES). Users have the option to manually override this.

## 2.3 Usability, Transparency & Explainability in Scientific Tools

Successful beginner-oriented scientific tools minimise cognitive load and support learnability. Classic HCI guidance (e.g. Nielsen’s usability heuristics) recommends consistent layouts, clear labels, and instant feedback (Amershi, 2019). This enhances the ability for new users to quickly form correct mental models. ISO 9241-110 (interaction principles) and ISO/IEC 25010 (software quality model) standards frame verifiable qualities for software design. In relation to this, the GUI uses progressive disclosure (tutorials before parameter dialogs), descriptive tooltips for parameter choices, and a visible status bar for success/error status. This is aimed at lowering the barrier to entry for first-time users.

Both transparency and explainability are equally important in scientific tools (Neilsen, 1994). Users should be able to see exactly what was run and why the outcomes look the way they do so analyses can be trusted and reproduced. Good practice should focus on exposing parameters, explaining assumptions, and a complete account of settings so results can be verified or reproduced later (Wilson, 2017). Explainability provides answers to users about why they have the findings they do. The GUI annotates edge semantics (e.g. -->) and includes an explanation via separating sets (Bareinboim, 2016), plus an algorithm comparison tab (contrasting PC, GFCI and FGES outcomes). These features map to beginner friendly guidance required by an explainable system (International Organisation for Standardization, 2020). This provides rationale, points out uncertainty/limitations and supports comparison, while keeping simple interactions for novice users (Nielsen, 1992). Together, these design choices support this project’s aim of developing a beginner-friendly GUI which guides and teaches new users and exposes core causal assumptions.

## 2.4 Design Principles

This GUI is guided by a set of evidence-based principles for beginner users. Firstly, progressive disclosure and clear labelling prioritise learnability: tutorial wizards explain ideas before parameter dialogs and use plain language for users to recognise rather than recall (International Organisation for Standardization, 2020). Secondly, visibility of system status and consistent, predictable interfaces. Parameters are saved in exports and standard default settings run on typically formatted datasets (International Organization for Standardization, 2023). Thirdly, built in guard rails enable error prevention and graceful recovery (e.g. Graphviz failure -> spring-layout fallback)(International Organisation for Standardization, 2020). Fourthly, transparency and provenance support trust and reproducibility, important attributes of scientific tools (International Organization for Standardization, 2023). Finally, explainability and accessibility are addressed via edge-mark semantic explanations, “Why not connected?” tab, and a cross-algorithm comparison tab. High-contrast visuals and consistent layouts also aid with accessibility (Palmer, 2025).

## 2.5 TETRAD & py-tetrad

TETRAD is a well-established, open-source toolkit for causal discovery and causal inference (developed at Carnegie Mellon University)(Ramsey, 2018). It implements the different search algorithms used in this GUI (PC, FGES, GFCI)

Py-tetrad is a Python bridge to TETRAD’s Java engine (Carnegie Mellon University, 2023). It initiates a Java Virtual Machine (JVM) to expose Python wrappers for datasets, test/scores and causal searches. This product uses a thin adapter (TetradRunner) to hide the Java functionality. The GUI passes a pandas.DataFrame and parameters to the runner which returns a NetworkX graph. The graph is annotated with endpoint metadata and is rendered and explained by the GUI.

A small set of parameters are standardised across algorithms in the GUI:

* PC/GFCI: α (significance value), search depth (-1 for unlimited), and dtype (auto/continuous/discrete) to select the correct test (Fisher-Z or χ²/G²).
* FGES: penalty\_discount (BIC scaling) and dtype to select the correct score

At startup, the application checks for a suitable JDK for TETRAD as it runs inside the JVM. If missing, clear setup guidance is reported. One JVM instance is kept per session, and calls are safe guarded with try/except to catch Java exceptions and report with user-friendly dialogs.

Finally, TETRAD’s graph outputs are normalised with consistent edge endpoints (e.g. -->, ---, o->, o-o)(Carnegie Mellon University, 2025). These are attached to edges in the NetworkX graph to be used in the Graph View, Edge Explanation and Compare tabs. (Endpoint mark semantics are explained next in section 2.7.)

## 2.6 Graph Types & Edge Marks

**The kinds of graphs shown by the GUI:** causal discovery returns a pattern rather than a “true” DAG. Different mark semantics are used depending on the type of search algorithm that was run:

* DAG (Directed Acyclic Graph): each edge has a single direction (e.g. A --> B). Usually, observational data alone is not enough for this to be the direct output in causal discovery.
* CPDAG (Completed Partially Directed Acyclic Graph): An equivalence class of DAGs that matches the assumptions and data. Allows undirected edges where the direction of causality can’t be determined purely by the observational data. PC and FGES both output CPDAGs (no latent variables are assumed) (Spirtes, 2001)(Chickering, 2002).
* PAG (Partial Ancestral Graph) An equivalence class that allows latent confounders. PAGs can have regular arrowhead endpoints as well as circles (for unknown relationships)(Richardson, 2002). GFCI outputs PAGs (Ogarrio, 2016).

This GUI normalises the endpoints into a readable set of edge marks on the rendered graphs and in the right-hand edge list tab:

* A --> B (directed): A is a cause of B (possibly an indirect cause) within the equivalence class. This rules out the idea that B causes A but can’t guarantee there are no hidden common causes between the two variables.
* A --- B (undirected): A and B are adjacent in the equivalence class, but orientation can’t be determined purely from observational data. CPDAG can output this relationship.
* A o-> B (circle to arrow): B is an effect, but A’s status cannot be resolved. Could either mean A causes B or A and B share a hidden cause or both.
* A o-o B: (circle to circle): A and B are adjacent but with high uncertainty. The direction of the relationship or possibility of confounding variables is not resolved purely by the data or assumptions of the search.

**How to read these in context:**

* PC/FGES (CPDAG): consists of --> and --- relationships. Undirected --- indicates the edge exists but no direction can be determined by the DAG in that equivalence class (Andersson, 1997).
* GFCI (PAG): consists of --> and o-> and o-o. Can detect hidden confounding variables and allows uncertainty.

**What the marks can’t tell us:** they do not encode the size or linearity of the causal effects. They are conditional on the Markov and faithfulness assumptions as well as the test/score used. Arrows express claims that are true in all graphs in the equivalence class (Pearl, 2000), which is why circles are used to show varying endpoints across valid graphs.

**Where the GUI uses this:** the edge explanations require these endpoint marks (to click an edge in the list), and the comparison tab to report orientation and type disagreements between the three implemented algorithms. The “Why not connected?” tab uses endpoint marks to report a separating set for missing edges (Spirtes, 2001).

# ***Chapter 3: Project Specification***

## 3.1 System Overview

This project delivers a GUI for causal discovery designed to make running and interpreting standard causal discovery algorithms accessible to students and novice researchers. My application guides users from CSV import -> deciding algorithm (PC, GFCI or FGES) -> visualisation of graph -> graph explanation -> export, emphasising education, clarity and reproducibility of generated results.

The core workflow involves a left-hand “Tools” panel providing actions: “Import Data”, “PC Search”, “GFCI Search”, “FGES Search” and a top-hand tool bar with actions: “Help”, “Tutorials”, “Export Graph”, “Export Edges”, and “Export Comparisons”. The central workspace consists of a stacked view with a Data Preview canvas which confirms the loaded CSV, and a Graph View which displays generated DAGs. Searches call py-tetrad on the backend and accept user inputted parameters via dialogs (e.g. depth, data type). Results are drawn by converting to a NetworkX graph.

Learned graphs display labelled nodes with edge endpoint styles matching conventional PAG/CPDAG rules. The layout uses Graphviz “dot” if available or a fallback to retry with safe-IDs. Nodes can be dragged by users to customise the DAG’s layout. The canvas summarises parameters used in the graph’s subtitle for reproducibility.

The right-hand panel contains an “Edges” tab which explains the selected edge type in plain language, a “Why not connected?” tab showing reporting separating sets which explain missing edges via conditional independence testing, and a “Compare” tab that runs the other algorithms on the same data/parameters and reports unique differences.

A “learn by doing” approach is utilised with tutorial wizards and concise tooltips for beginner users.

Users can export graphs as PNGs, edge lists as TXTs and comparison reports as TXTs. A status bar reports progress and error messages allowing for a stable experience. The software runs offline preventing any chance of data protection issues.

## 3.2 End Users

This GUI targets three groups:

1. Students who are learning causal discovery for the first time
2. Novice researchers and analysts applying PC, GFCI, and FGES searches to their own datasets
3. Instructors/teaching staff demonstrating causal discovery algorithms in classes

All users access the same features (accounts and roles are unnecessary): loading CSVs, running searches, exporting graphs/edges/comparisons, and using tutorials and explanations to interpret results.

## 3.3 Functional Requirements

The functional requirements are written using the MoSCoW prioritisation format (Agile Business Consortium, 2025) and are organised into eight categories that relate to the interface’s key areas. Structuring the functional requirements this way supports testing and evaluation steps.

***General***

1. The system **must** launch and display the main window
2. The system **must** display a tool panel listing “Import Data”, “PC Search”, “GFCI Search”, and “FGES Search”
3. The system **must** present a central workspace with a “Graph View” widget
4. The system **must** show a status bar to report progress and error messages
5. The system **could** offer a light/dark theme in the menu and remember the selection
6. The system **could** launch as a single packaged executable

***Data handling***

1. The system **must** open an OS native file picker when clicking “Import Data”
2. The system **must** accept CSV files and load them into a pandas DataFrame
3. The system **should** show a small data preview of the first N rows of the CSV in the data page
4. The system **should** warn on clear errors in the CSV (e.g. missing headers)
5. The system **could** warn on extremely large datasets (e.g. >10,000 variables)
6. The system **could** support drag and drop CSV into the window

***Algorithm execution***

1. The system **must** run PC with the users inputted alpha, depth and datatype in the parameter wizard
2. The system **must** run GFCI with the users inputted alpha, depth and datatype in the parameter wizard
3. The system **must** run FGES with the users inputted penalty discount and datatype in the parameter wizard
4. The system **must** convert Tetrad output to a NetworkX graph that has correct edge endpoint marks
5. The system **must** pass parameter choices from the parameter dialog and correctly pass them to py-tetrad
6. The system **should** capture the correct data type/test/score used using auto detection and display in the graphs subtitle
7. The system **could** cache the user’s previous result from Tetrad so that layout changes do not re-run the algorithm

***Visualisation***

1. The system **must** render nodes as rounded boxes and render edges with the correct endpoint markers (e.g. arrows, circle and undirected)
2. The system **must** use Graphviz dot layout if it is available with a safe-ID fallback option
3. The system **must** allow repositioning of nodes by clicking and dragging and for edges to be updated
4. The system **should** avoid overlap of variable labels and edge endpoints
5. The system **could** support zooming and panning via mouse
6. The system **could** highlight a selected node and its connected neighbours

***Explainability & Comparison***

1. The system **must** provide a “Why not connected?” tab that attempts to find a separating set using conditional independence testing to interpret why an edge does not exist
2. The system **must** provide an “Edges” tab which lists all edges and their types. Clicking an item explains that edges relationship
3. The system **must** provide a “Compare” tab which runs PC, GFCI and FGES on the same data using the same parameters and lists all the disagreeing edges between the algorithms
4. The system **should** display the parameters at the top of the comparison tab
5. The system **could** allow you to switch between viewing graphs from each of the PC, GFCI and FGES algorithms

***Usability & Learning Aids***

1. The system **must** include tutorial wizards for “Import”, “PC’, “GFCI”, “FGES” with “don’t show again” options
2. The system **must** include a “Tutorials” hub on the toolbar where you can launch any of the tutorials at any time
3. The system **should** add concise tooltips to tool items and parameter fields to explain their meaning
4. The system **could** add a first run tutorial overlay of the UI for new users

***Export & Reproducibility***

1. The system **must** export the rendered graph to PNG with the title and subtitles
2. The system **must** export the graph’s edge list and the current comparison report with markers to TXT
3. The system **should** stamp exports with the algorithm’s name and parameters used
4. The system **could** export edges to CSV as well as TXT

***Error Handling & Robustness***

1. The system **must** show an error message if a search is requested with no loaded data
2. The system **must** catch exceptions from py-tetrad and show an error message without crashing the UI
3. The system **must** handle the absence or failure of Graphviz using the fallback methods
4. The system **should** write exceptions and layout decisions to a log and the status bar for troubleshooting purposes
5. The system **could** validate parameter ranges with a pop-up message

## 3.4 Non-Functional Requirements

The non-functional requirements are written using the MoSCoW system and specify quality attributes the GUI should exhibit under normal use.

***Performance***

1. The system **must** launch and render the main window in < 1.5 seconds on a mid-range desktop (8 GB RAM)
2. The system **must** respond to UI actions in < 300 milliseconds (not including algorithm runtime)
3. The system **must** export a DAG to PNG in < 500 milliseconds after clicking save in the user’s native file explorer
4. The system **should** complete PC Fisher-Z test with default parameters on a 2,000 x 12 CSV in < 1 second on the test machine

***Compatibility***

1. The system **must** run on Python 3.9-3.12 on macOS 12+

***Usability & Accessibility***

1. The system **must** let a first-time user load a CSV, run PC, GFCI and FGES algorithms, and view a DAG within 5 minutes without external help

***Reliability***

1. The system **must** handle invalid CSVs, missing Java or py-tetrad, or any layout errors with no crashes
2. The system **must** preserve loaded data and chosen parameters when resizing windows or switching between tabs
3. The system **should** timestamp errors in a log txt file

***Security***

1. The system **must** run entirely offline to keep all user data local

***Maintainability***

1. The system **must** keep the UI, controller, and Tetrad interface modules separate
2. The system **should** centralise the algorithm parameter mapping for the PC, GFCI and FGES algorithms in one place

***Portability & Build***

1. The system **should** provide clear guidance for setting up when Java or py-tetrad is missing with start instructions

## 3.5 Datasets

To validate the GUI end to end, I used three public datasets as a benchmark with well-known structures, with small numbers of variables for fast, interactive test runs:

* LUCAS (Lung Cancer Simple) discrete, 12 variables. Used to check sensible DAGs, Edge explanations and “Why not connected?” tab, and exports
* ASIA (Chest Clinic) discrete, 8 variables
* Sachs (protein signalling) continuous, 11 variables

These datasets checked sensible DAGs, edge explanations, the “Why not connected?” tab, the “Compare” tab, exports, and contrast score vs constraint-based searches.

All datasets were loaded as CSVs with original variable names. No personal data were involved. Minimal preprocessing was applied to consistently test all datasets as CSVs.

## 3.6 Constraints & Risks

Development of this GUI was done using a 2022 MacBook Air (M2, 8GB RAM) running macOS Ventura 13.2. The software depends on Python (3.9-3.12), PyQt5, Matplotlib, NetworkX, py-tetrad, and Graphviz. All software is open source, and literature is accessed with open access or University of Birmingham institutional login. Key constraints/risks are: i) hardware limits which could slow algorithm runtime and exhaust memory on large datasets, and ii) third-party dependencies (Java/Graphviz availability, version mismatches). All datasets used are public and data is processed locally. Licensing is discussed later in this report.

## 3.7 Success Criteria & Evaluation Plan

**Success criteria.** The GUI is completed when a new user can load a CSV, run PC, GFCI or FGES on their data, view and interpret the DAG, interpret edge explanations, use the “Why not connected?” tab, and export DAG PNG, edge lists, and comparisons within 5 minutes, unassisted. With LUCAS, ASIA, and Sachs, all must produce a graph and comparison list for each algorithm. The app should launch in under 1.5 seconds and respond to UI actions in under 200 milliseconds. Invalid CSVs or missing Graphviz should not cause crashes, and exports should match the displayed graph and parameters. The codebase should separate the Qt UI, controller, and Tetrad interface.

**Evaluation plan.** Verification will use a functional test matrix with LUCAS/ASIA/Sachs datasets to follow the full workflow, with pass and fail notes. The launch and UI latency and a PC search will be benchmarked on the reference machine. A quick usability spot test (3 novices) will time the core workflow and collect brief task ease ratings. Robustness checks will remove Graphviz to confirm stability. Export validation will compare PNG and TXT outputs to the current canvas and edge list. An accessibility audit will check colour contrast and text size in tutorial wizards and pop-ups.

# ***Chapter 4: Design***

This chapter outlines the design choices of the causal discovery GUI. It covers the architecture and components, the data/control flow and integration of algorithms (PC, GFCI, FGES), visualisation and UX design, and robustness and performance. These sections cover the practical blueprint that was implemented in development and frames the evaluation later in this report.

## 4.1 Architecture & Components

**Architecture.** The application is an offline desktop app which runs from a single process with a light layered MVC structure: **Presentation** uses Qt widgets and tutorial wizards, **Controller** uses interaction logic, caching results and error handling, **Algorithm module** bridges to py-tetrad, and a **visualisation** module uses NetworkX and Matplotlib. This ensures responsibilities are separated without complex services or a need for a database. This is an appropriate design for this application to run entirely on a student laptop with low latency.

**Key components.**

* **UI (presentation).** A “MainWindow” built using Qt Designer holds the “Tool panel” (Import Data, PC, GFCI, FGES), the “Tool Bar” (Help, Tutorials, Export Graph, Export Edges, Export Comparisons), a central “Graph” tab, “Data Preview” tab, and right-hand tabs (Edges, Why not connected?, Compare). New users are guided by wizard tutorial dialogs (PcWizard, GfciWizard, FgesWizard) and parameter dialogs (GfciParamDialog, FgesParamDialog). EdgeExplanationDialog formats explanations of edge types.
* **Controller.** Controller connects user inputs to functions. It loads CSVs into panda dataframes, runs searches, draws graphs, populates side panels, builds comparisons, and exports PNGs and TXTs. It also shows actionable messages on errors.
* **Algorithm bridge.** TetradRunner wraps py-tetrad/Java calls for PC, GFCI, and FGES searches, maps GUI parameters to Tetrad and returns NetworkX graphs with correct endpoint markers (-->, ---, o->, o-o). A helper exposes a Python sepset finder for the “Why not connected?” tab to explain missing edges.
* **Visualisation.** The controller sends a Matplotlib canvas to Qt to draw variable nodes as rounded rectangles and edges with correct arrowheads for each endpoint. Graphviz “dot” layout is preferred if available and a fallback is in place to a spring layout. Users can click and drag nodes to customise DAG layouts. Redraws keep interaction with the graph smooth.

**Caching & data flow.** “Import Data” reads the CSV into a DataFrame and invalidates caches. Running PC/GFCI/FGES has the controller call TetradRunner and draws the result, populates the “Edges” list and caches information for the other algorithms. A results cache is maintained holding algorithm, data, and parameters and a comparison cache is maintained holding data and parameters. Text is generated for the “Compare” tab once and gets reused when exporting comparisons to avoid re-running algorithms if data and parameters are unchanged.

**Robustness & fallbacks.** When Graphviz is missing there is an automatic fallback layout. If Java/py-tetrad or CSV parsing fails, the user sees a clear error message and no crash. Loaded data and parameters are remembered when switching tabs or resizing the UI, and graphs are exported as seen on the GUI.

**Extensibility.** New algorithms are easily added using the same py-tetrad bridge along with a new tutorial wizard and parameters dialog.

## 4.2 Data & Control Flow & Algorithms

**Data path:** Tool panel allows users to import their CSV. The file is loaded into a pandas DataFrame, basic checks are run on headers and quantity, and a lightweight cache of filename, rows, and column names is recorded. The “dtype” setting (auto/continuous/discrete) is passed to statistical tests. Auto aims to infer the correct py-tetrad test from the inputted data.

**Control flow:** UI actions are handled by the Controller: i) read CSV and invalidate caches, ii) open a tutorial wizard and parameter dialog then call “TetradRunner”, iii) convert to a NetworkX graph with endpoint annotated edges and render with Matplotlib, iv) populate the edge list and node pickers in the “Why not connected?” tab, and v) enable exports.

**Algorithms.** TetradRunner is the algorithm bridge which wraps py-tetrad for PC, GFCI, and FGES. GUI parameters are mapped to Tetrad, and the search algorithm is executed returning a standardised graph. PC and GFCI use constraint-based conditional independence testing (α, depth). FGES uses score-based search with the penalty discount parameter. The “Why not connected?” tab uses a Python sepset finder. The “Compare” tab generates edge sets for each algorithm and reports unique edges and disagreements between searches and saves current parameters in the header for reproducibility.

## 4.3 Visualisation & UX

This GUI has a high contrast graph view (black, white, and grey with navy accents). Rounded rectangles display for nodes and specific endpoint marks are used for edges (-->, ---, o->, o-o). A clean dot layout using Graphviz is prioritised over a reproducible spring layout if unavailable. Graph layout can be tidied and customised by the user. Custom positions are preserved when running another search. A status bar relays progress and error messages, and exports are one click buttons for quick tasks.

The right panel shows an “Edges” list mirroring what is shown in the DAG. Any row clicked opens an edge explanation pop-up with a navy heading highlighting the selected edge (e.g. A o-> B), followed by “Meaning:” in bold with a regular weighted body of text explaining the edge. A & B are substituted for the selected variable names. Text can be selected and copied.

Two tabs are built in to aid the user’s understanding. “Why not connected?” tab allows selection of two variables and a short reason behind the two variables lacking an edge if a separating set S is found. “Compare” tab shows search algorithm comparisons generated from cached graphs keeping a low-latency experience.

Overall usability is guided by clarity and jargon is kept to a minimum. Key usability features are tooltips on all major controls, tutorials shown before running searches, consistent labels, and no reliance on colour allowing shapes and text to convey meaning. Minimal colour palette of black, white, grey and navy with appropriate font size enhances readability and gives all dialogs and tutorial wizards a consistent visual style.

## 4.4 Robustness, Performance & Dependencies

**Robustness:** the GUI blocks common failure modes and avoids crashes. File I/O and search algorithm calls are embedded in try/excepts with understandable dialogs. If Graphviz dot is not found a fallback to a spring layout is used for graphs meaning a DAG is always rendered. Explanations substitute A/B variables with HTML, so pop-ups match the list item clicked by the user. Graphs and comparison text get cached and are invalidated when changing data or parameters. Exporting graphs and comparisons read information from cache. UI state (position of nodes, window size, parameter choices) is preserved.

**Performance:** minimal startup work and progress shown by status bar. All algorithm’s graphs are fetched and cached when a search completes. The “Compare” tab and “Export Comparisons” tool read from cache rather than re-running all search algorithms. If Graphviz fails, the seeded spring layout generates fast and reproducible node positions. The “Why not connected?” tab searches for sepsets with a small conditioning set size (e.g. 3) for efficient interactivity. This keeps interactions snappy and practical even on mid-range hardware.

**Dependencies:** the app runs completely offline and uses local libraries:

* Core: Python 3.9-3.12, PyQt5, Pandas, NumPy, NetworkX, Matplotlib
* Causal engine: tetrad\_iface (Python wrapper) and Tetrad Java .jar (tested with OpenJDK 21)
* Optional Graphviz and pydot for “dot” layouts

All dependencies used are open source with the versions documented in “requirements.txt”. the Java/Tetrad versions are found in the README). If Java/Tetrad or Graphviz are missing, errors are caught and explained with a setup guide, rather than the application crashing.

# ***Chapter 5: Project Management***

## 5.1 Development Methodology

The project used a lightweight Kanban approach suited to the evolving requirements of the application. Tasks were tracked on a simple board consisting of backlog, in progress, verify, and done. This had strict work in progress limits to maintain a productive flow. Each iteration delivered a small, testable feature which was quickly checked with integration testing ensuring the application was always in a usable state. The task backlog implemented MoSCoW prioritisation. Should/could items were only pulled when capacity allowed. The continuous flow of this development strategy led to steady progress towards the requirements defined in Chapter 4, without the need for sprints.

## 5.2 Timeline

To keep steady development of the application, I planned the project with a lightweight Gantt chart (Figure 5.2) alongside the Kanban workflow. The chart maps the summer timeline across the main phases of the project (literature review, design, GUI + py-tetrad integration, tutorials/explainability, exports, testing, report writing and final submission). The tasks were sequenced in a sensible order to reduce risk in development (e.g. py-tetrad algorithms before DAG rendering) and some tasks intentionally overlap when necessary (e.g. user testing during the report writing phase). The Gantt chart was an adjustable schedule functioning as a baseline progress tracker to help pull work across the Kanban board. This allowed for re-prioritisation if issues arose and ensured delivery of a working application within the project deadline.

## A graph with blue squares AI-generated content may be incorrect.

*Figure 5.2: Gantt chart*

## 5.3 Risk Management

Due to this project mostly being developed on a single personal laptop, the main risk to manage is hardware failure or corruption of the application’s files. To avoid any extreme loss of progress, all files related to this project were pushed to GitHub at regular intervals and backed up on a personal OneDrive folder, enabling recovery. This also allowed progress to be made on any machine when access to the main laptop used was not possible, contributing to efficient progress. No other major risks were identified.

# ***Chapter 6: Implementation***

## 6.1 Technologies Used

The GUI desktop interface is implemented in Python with PyQt5. CSV loading uses pandas. DAG modelling and rendering uses NetworkX and Matplotlib. Running search algorithms uses the Java Tetrad library locally. Optional high-quality layouts use Graphviz then a spring layout to fall back on, keeping the app robust. A lightweight QSS stylesheet is used for consistent theming. This stack is effective for efficient cross platform development. Python and PyQt allow fast iteration and native desktop feel, and py-tetrad ensures reliable causal discovery rather than re-implementing algorithms.

## 6.2 Modules & Key Files

The application’s codebase is small and modular with each file having a clear specific role.

* **app.py:** Creates “QApplication”, loads “ui/main\_window.ui”, applies “styles.qss”, creates a “Controller” instance, connects UI functionality (tool panel, exporting, tutorial wizards) and surfaces environment checks for Java and Graphviz using the status bar and dialogs.
* **controllers.py:** manages the user facing aspects.
  + state & parameters: holds loaded DataFrame, alpha/depth/dtype/penalty\_discount, the current networkx.DiGraph, and the last\_algo
  + caches: \_result\_cache (per-algo graph), \_compare\_cache (comparison text) and \_invalidate\_caches() on data/parameter changes
  + runs: run\_algorithm(algo) calls TetradRunner.run\_search(), populates edge list and node pickers, then builds the comparison tab using caches
  + visualisation: draw\_graph() renders nodes as rounded rectangular labels and endpoint marks with Matplotlib. Hover and dragging is supported. \_graphviz\_pos() uses Graphviz “dot” or spring fallback
  + explainability: populate\_edge\_list(), \_parse\_edge\_label(), show\_edge\_explanation() opens the explanatory dialog and lists edges (-->, ---, o->, o-o)
  + why not connected: separating sets and missing edge reports are generated with explain\_non\_edge()
  + comparison & exports: \_build\_comparison\_text(), export\_graph\_png(), export\_edges\_txt(), export\_comparisons\_txt() run the text exports for nodes, edges and parameters
  + error handling: exceptions get converted into actionable dialogs the status bar displays
* **dialogs.py:** modal UI.
  + Tutorial wizards: “ImportWizard”, “PcWizard”, “GfciWizard”, “FgesWizard” and “TutorialsHub”. “QSettings” remembers the “Don’t show again” preference
  + Parameter dialogs: “GfciParamDialog” (also used for PC with a renamed header), “FgesParamDialog”
  + Edge explanations: “EdgeExplanationDialog” renders navy heading and bold text with HTML, and the green styled Continue button with QSS.
* **tetrad\_iface.py:** bridge to py-tetrad.
  + Load\_dataframe(path)
  + Run\_search(df, algo, alpha, depth, dtype, penalty\_discount) generates networkx.DiGraph with edge attributes and endpoints (-->, ---, o->, o-o)
  + Find\_sepset\_py() is called for the edge explanation tab.
* **styles.qss:** Styles the application using neutral black/white and grey with navy accents. This also styles lists, dialogs, and green continue buttons for tutorial wizards.
* **ui/main\_window.ui:** Tool panel with “Import Data”, “PC Search”, “GFCI Search”, “FGES Search” and tool bar with “Help”, “Tutorials”, “Export Graph”, “Export Edges”, “Export Comparisons”. Right hand “Edges”, “Why not connected?”, and “Compare” tabs and a status bar

## 6.3 Algorithm Integration & Edge Mark Rendering

**Integration:** when clicking either PC, GFCI, or FGES search, TetradRunner.run\_search(df, algo, depth, dtype, penalty\_discount) is called via the controller. Py-tetrad is configured with the constraint test, score and given parameters. A networkx.DiGraph is returned. Edges are annotated with:

* “mark” (-->, ---, o->, o-o) for CPDAG and PAG edge meanings
* Endpoint flags “ep1/ep2” (ARROW, CIRCLE) used for graph drawing

**Algorithm specifics:**

* **PC** (constraint-based algorithm): For continuous data, Fisher-Z is used. For Discrete data χ²/G² is used. Users input desired alpha and depth parameters (unlimited when depth = -1). CPDAGs are usually outputted with directed (-->) and undirected (---) edges.
* **GFCI** (hybrid): Uses a combination of independence testing and scoring. Outputs PAG style marks (o->, o-o) to represent latent confounding variables and orientation uncertainty.
* **FGES** (score-based): maximises a data appropriate score (e.g. BIC/BDeu). CPDAGs are outputted with directed (-->) and undirected (---) edges. The penalty\_discount parameter is recorded with results

**Parameter mapping & dtype:** For PC and GFCI, user inputted alpha, depth, and dtype (cont/disc/auto) are passed to py-tetrad. For FGES the user inputs penalty\_discount and dtype. For auto dtype, the bridge infers the dtype from the DataFrame and picks the most appropriate test/score. Py-tetrad errors are caught, and an actionable dialog is displayed.

**Caching & reproducibility:** Results are cached (algo, data, parameters, comparison text) so any re-rendering or file exports does not cause re-runs of search algorithms. Nodes, parameters, and edge lists are exportable for reproducibility.

**Mark normalisation:** Consistency across lists, pop-ups, and drawings is ensured by edge labels (e.g. -> and 🡪) being normalised to -->. The edge list on the right panel uses the exact same “mark” strings so the edge markings are consistent between the drawn graph and edge explanation tabs.

**Rendering model:** the controller converts “mark” to endpoint symbols:

* --> = ARROW head
* --- = undirected, no endpoints
* o-> = CIRCLE tail, ARROW head
* o-o = CIRCLE tail, CIRCLE head

First a base line is drawn (---), then circles and arrowheads are overlayed. Arrowheads use Matplotlib annotations with readable sizes for accessibility.

**Layout & readability:** When Graphviz “dot” is available its layout is preferred. If unavailable, a seeded spring layout calculates stable node positions instead. Node labels are displayed with rounded rectangles and are draggable to reposition. Live edge re-computation maintains clear attachment when moving nodes

**Explainability link:** When an edge in the edge list is clicked, the “Edge Explanation” dialog opens. The title “A mark B” replaces A and B with selected variable names and mark with the specified end point. A concise interpretation of the selected edge meaning is given below.

This design keeps a consistent presentation of the orientation calculated by py-tetrad across lists, pop-ups, exports and the rendered DAG.

## 6.4 Comparison Generation & Export

**Workflow:** Once a search algorithm has run, the controller generates a cross-algorithm report with \_build\_comparison\_text(). \_get\_or\_run\_graph() computes all three graphs for PC, GFCI, and FGES and constructs lists of edges and their endpoint marks. Parameters (alpha, depth, dtype, FGES penalty) and algorithm specific edge counts are recorded in the header. The report body contains: i) algorithm unique edges (present in only one search result) and ii) orientation and type disagreements when variable pairs differ in direction or endpoint mark between algorithms. Stable sorting of results displayed in the “Compare” panel improves interpretability.

**Caching & Performance:** Comparison text is cached so that opening or exporting the “Compare” tab reuses cached text and graphs. This prevents re-running of algorithms, so “Export Comparisons” is instant. Caches are invalidated if data or parameters are changed.

**Exports:** Three export actions:

1. **Export Graph:** saves PNG of the current rendered graph from Matplotlib at 300 dpi.
2. **Export Edges:** saves TXT of the numbered edge list with the run parameters.
3. **Export Comparisons:** saves TXT of the exact comparison block in the UI’s “Compare” tab (header, counts, unique edges, disagreeing edges) plus the DAG title / CSV filename.

Sensible default filenames used by exports come from the DAG title or CSV filename if untitled. Confirmations of exports are shown by the status bar. Combining clear headings, cached generation and endpoint mark consistency allows a reproducible way to interpret the differences between PC, GFCI, and FGES searches on the same dataset.

## 6.5 Tutorial Wizards

**Purpose:** Tutorials reduce parameter anxiety (the selection burden novice users experience) making the GUI usable for beginners to causality. The GUI has lightweight, context sensitive tutorials for “Import Data”, “PC Search”, “GFCI Search”, and “FGES Search”, and a central “Tutorials Hub”. Each tutorial wizard covers when to use the algorithm, required assumptions (e.g. data type, independence tests, scores), important parameters (e.g. depth, penalty discount) and understanding the output (edge marks). Wizards finish with a concise list of advantages and disadvantages so users can select the most appropriate algorithm with confidence.

**Launch & persistence:** Wizards open automatically when a tool is used and can be revisited from the “Tutorials Hub”. Users can toggle “Don’t show again” which persist via QSettings (show\_pc\_qizard, show\_gfci\_wizard, show\_fges\_wizard) to avoid tutorials interrupting. Cancelling a tutorial for each search algorithm proceeds to the parameter dialog.

**Implementation:** dialogs.py houses tutorial wizard dialogs. They are styled to match the app’s consistent theming and use a green continue button as other instructional pop-ups do. HTML based pages embed the content and figures. The “Tutorials Hub” is a selection list which launches tutorial wizards and returns the selected title.

**Maintenance & extensibility:** Copy on template pages allows for new topics to be implemented in the future without changing the controller. Tutorial wizards do not change state beyond the “Don’t show again” option keeping them low risk and easy to be updated as new tools are added.

# ***Chapter 7: Testing***

## 7.1 Unit Tests

I have used unit testing to verify that small functions in the code work as intended in isolation. I used pytest and log capture to focus on testing the most critical functionality to reliability and responsiveness in the GUI. One unit test simulated the absence of Graphviz by having the \_graphviz\_pos() function return None. This triggers the documented fallback to nx.spring\_layout without crashing and outputs an understandable debug message to the console. The second unit test validated the result-caching path that prevent redundant re-runs of search algorithms. \_df\_signature, \_param\_signature, and \_graph\_key were checked for stability (same data and parameters = same key) and sensitivity (changing parameters or data = different key). When \_invalidate\_caches() runs, the call sequence displays “CACHE MISS -> run\_search() -> CACHE HIT” in the captured test logs.

These tests provide reliable pass/fail signals when testing edge cases early (e.g. missing dependencies or cache). This directly supports the non-functional goals including reliability, responsiveness, and maintainability.

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 7.1.1: Console logs from unit tests showing cache miss/hit/invalidate outputs and the “not importable -> fallback” message when Graphviz is not found.*

## 7.2 Integration Tests

Integration testing was used to verify that UI events, controller logic, the Tetrad runner, and render/export worked end to end as each feature was added. The user flow was tested for each new component as it was added and verified it was working as intended. For example, when the “FGES Search” button was added, tests ensured it opened the tutorial wizard pop-up, and then the parameter dialog. Tests were done to check pressing run produced and rendered a DAG in the central UI canvas and that the edge list was populated with the correct causal relationships.

Exports were also tested and verified. “Export Graph” opens the native file explorer and saves the current DAG layout to PNG. “Export Edges” saves the list of edges and their marks to a text file. “Export Comparisons” matches and saves what is displayed in the GUI’s “Compare” tab to a text file without re-running search algorithms. Guard rails were also tested. For example, clicking “Run” with no data was tested to confirm a warning is displayed. All these checks confirmed the modules interconnect as intended under both normal and erroneous conditions.

## 7.3 UI/UX Testing

I ran a brief task-based study with 7 BSc/MSc students who all reported no prior experience with causal discovery. After a 1-minute primer explanation, participants were instructed to complete the test workflow: Import CSV -> choose search algorithm -> set parameters -> run search - > view DAG -> export DAG to PNG. Afterwards, each answered 5-point Likert items (1=strongly disagree… 5=strongly agree). Results reported were consistently positive. Ease of CSV import (mean = 4.71), tutorial guidance (4.71), UI responsiveness (4.86), aesthetics/minimal clutter (4.71), completing the user test workflow in under 5 minutes (4.71), ease of export (4.57), and overall application’s ease of use (4.86). Perceived mental effort averaged a score of 3.71 (moderate load), which is understandable for first time users unfamiliar with the concepts of causal tools. These results provide direct support for the usability NFRs. Particularly that a first-time user can complete the full workflow in under 5 minutes without needing external help.

Outcomes suggested a small improvement. The Export actions should be more prominent/central (one participant rated neutral on the ease of exports). The most obvious limitations of the conducted user testing are: i) small, convenience-based sample of participants, ii) all tests ran on a single machine, and iii) measures are self-reported. Nevertheless, these results support the GUI being intuitive, responsive, and beginner-friendly and aligns with the project’s aims.

## 7.4 Performance & Responsiveness

The performance of three different functionalities was measured using performance counters as a timer: application launch, exporting DAG to PNG (from clicking save to completion of saving PNG), and PC execution on the LUCAS dataset (2000x12). Each test was run three times on a MacBook Air M2 (8GB RAM, macOS 13.2). Figure 7.4.1 reports all three runs and the mean for each feature. App launch took on average 190 ms (181 – 196), exporting a DAG to PNG took on average 159 ms (154 – 163), and PC search took on average 841 ms (805 – 895). These satisfy the non-functional requirements stated in section 4.4 of this report: app launch < 1.5 s, DAG to PNG < 0.5 s, and PC on a 2000x12 dataset < 1 s. The variability in export time reflects first run warmup effects. Algorithm timings will vary with increased sample size of datasets, variable count, and parameters. Overall, the measurements confirm a responsive UI with large headroom around the stated non-functional requirements of the application.

A table with numbers and text

AI-generated content may be incorrect.

*Figure 7.4.1: per functionality timings (ms) for app launch, export DAG to PNG, and PC search (on 2000x12 dataset). Measured with three runs on a MacBook Air M2 (8GB RAM, macOS 13.2). The table reports the timings of all three runs and the mean for each feature.*

## 7.5 Summary

This chapter verified the correctness and quality of the GUI from the unit testing level up to the end-to-end use. Unit tests confirmed two reliability hotspots: i) working Graphviz fallback to spring layout with clear error logging, and ii) result caching of data/parameter keys to prevent redundant algorithm runs. Integration checks confirmed the complete application workflow: tutorial -> parameter dialog -> algorithm execution -> rendered DAG -> populate edge list -> exports, and validated guard rails (e.g. running searches with no data). The UI/UX study with 7 novice users reported high ease of use (4.86/5 overall), quick first run success (under 5 minutes), responsiveness, and an uncluttered design. User feedback highlights the need for making exporting more prominent. Performance measurements on a MacBook Air M2 showed average launch time of 190 ms, average PNG export time of 159 ms, and an average PC run on a 2000x12 dataset time of 841 ms. This comfortably meets the goals of the stated NFRs. Together, this all supports the project’s goals of a beginner friendly, reliable, and responsive causal discovery GUI, while also highlighting future work to broaden platform testing, using larger datasets, and improving error logging.

# ***Chapter 8: Evaluation***

## 8.1 Requirements Satisfaction

**Functional Requirements:**

Figure 8.1.1summarises the requirements pass/fail rates grouped into MoSCoW categories (must have, should have, could have). All Must requirements were met which cover the core workflow (“Import Data” -> set parameters -> run PC/GFCI/FGES -> render DAG -> explain/compare output -> exports). This includes the tool panel, central graph workspace, status bar, feedback, CSV import, parameter dialogs and correct integration to py-tetrad with robust error handling. Edge marks are normalised and visualised using Graphviz (or spring fallback). The right-hand panel houses the “Compare”, “Edges”, and “Why not connected?” tabs, and the tool bar contains export buttons (PNG, edge list to TXT, comparisons to TXT).

5/8 should items were delivered. Three were deferred to prioritise correct algorithm functionality, explainability and stability under the time constraints of this project: FR9) data preview of first N rows; FR23) no overlap of labels/endpoints; FR37) stamping exports with algorithm and parameter metadata (these details are reported inside the app via subtitles/compare tab). These features should be prioritised in future updates to the application.

2/11 could items were delivered. The deferred functional requirements were FR30) toggling between PC/GFCI/FGES graph views (compare tab already reports differences across algorithms) and FR38) edge export to CSV (TXT exports already implemented).

Overall, the pie charts show the prototype satisfies all critical capabilities for novice users to load data, run their choice of causal discovery algorithms with appropriate tests/scores, interpret outputted edges and absent relationships, compare outputs, and export results.

A green and red pie chart

AI-generated content may be incorrect.

*Figure 8.1.1: pie charts showing the number of functional requirements successfully met by my completed GUI application. These are categorised into must have, should have, and could have requirements.*

**Non-Functional Requirements:**

Figure 8.1.2 summarises the non-functional requirements (NFRs) categorised into MoSCoW prioritisation and status. Of the 9 **must have** NFRs, 8 passed testing and 1 is untested. Of the 4 **should have** NFRs, 2 passed testing and 2 did not (with no untested). This gives 10/13 passed, 2/13 unmet, and 1/13 untested requirements overall.

For the **must have** NFRs, performance and responsiveness targets were met when measured. Chapter 8.4 shows a mean launch time of 190 ms and PNG export time of 159 ms, both within the 1.5 s and 500 ms thresholds respectively. The NFR “UI action < 300 ms” remains untested due to time constraints for this project, though observed interactions appear instantaneous on the reference machine used for testing. First time user flow < 5 minutes, and robustness (handling of invalid CSVs, missing Java or py-tetrad, and Graphviz spring fallback) were all satisfied and discussed in Chapter 5 and Chapter 8 (unit/integration logs and error dialogs).

For the **should have** NFRs, two passed: PC search on a 2000x12 dataset completed in <1 s on the reference machine (Chapter 8.4), and centralised algorithm parameter mapping (PC/GFCI/FGES) for maintainability is evidenced in Chapter 7.3. Two NFRs were not met: timestamped error logging to a file and a first-run setup guide for missing Java or py-tetrad. Both were unimplemented to prioritise the core features ensuring usability and explainability and can be listed as minimal effort future enhancements.

*A green and red circles with white text

AI-generated content may be incorrect.*

*Figure 8.1.2: pie charts showing the number of non-functional requirements successfully met by my completed GUI application. These are categorised into must have and should have non-functional requirements.*

## 8.2 Usability Assessment

A study with 7 BSc/MSc participants assessed usability against the project’s design principles and NFR 6 (Section 3.4). Post test Likert responses were strongly positive (Section 7.3). These observations align with the intended beginner-friendly design. The tutorials reduced “parameter anxiety”, clear labelling assisted tool bar navigation, and the explanation tabs supported learnability. One neutral score points out the need for small UX refinements (more prominent Export actions). In summary, the study provides consistent evidence that the GUI is intuitive, response and supports novice users which satisfies the core usability requirements. Testing also revealed small areas for improvement on future iterations.

## 8.3 Case Studies / Example Analyses

**Datasets:**

* **LUCAS (Lung Cancer Simple set):** Synthetic, binary dataset for lung-cancer risk factors with a known causal Bayesian network. Features 12 binary variables with 2000 samples, ideal for testing a causal discovery method’s ability to recover the causal structure for reproducible demos.
* **Sachs (protein-signalling in human T-cells):** Real biological dataset that captures causal interactions between signalling molecules in human T cells. The dataset used in causal discovery has 11 variables with thousands of measurements. It is widely used as a benchmark in causal discovery as many edges are supported by real world experimental interventions
* **ASIA (Asia/Tuberculosis network):** A classic toy Bayesian network consisting of 8 variables (Asia travel, Tuberculosis, Smoking, Lung Cancer, Bronchitis, Either, X-ray, Dyspnea). It is commonly used for teaching and sanity checking causal discovery algorithms due to its small size allowing quick testing.

These three datasets jointly demonstrate the GUI is functioning correctly across small examples, synthetic benchmarks, and real-world intervention-based data.

To test that the GUI functions end-to-end as intended, searches were run on three representative datasets to capture the learned graphs, exports, and explanations.

**LUCAS (PC, α=0.05, depth=-1):** The GUI produced a 12 edge DAG with a readable layout showing clear endpoint marks (Figure 8.3.1 (A)). The exported edge list (with marks and run parameters) is shown in Figure 8.3.2. Running the comparison tab generated a concise report of all orientation/edge type disagreements across the three algorithms (Figure 8.3.3 (A)) and edges unique to one of the algorithms (Figure 8.3.3 (B)). GFCI often returned o-> when PC and FGES gave -->. This report was exported without re-running algorithms. The “Why not connected?” tab explained absences via conditional independence tests. For *Born an Even Day* with *Allergy* the sepset S was found with G² p = 0.1365, stating that the edge was not supported under the current run parameters (Figure 8.3.4). PNG and TXT exports preserved titles, parameters, and edge marks.

**Sachs (GFCI, α=0.05, depth=-1):** The result (Figure 8.3.1 (B)) showed a mixture of directed and partially oriented edges. This illustrates the GUI rendering endpoint shapes to communicate uncertainty of edges, a typical phenomenon in real biological datasets. Targeted “Why not connected?” queries aided interpretation of missing links, finding statistically supported conditional independencies.

**ASIA (FGES, penalty=2.0):** The small graph (Figure 8.3.1 (C)) served as a fast sanity check of the score-based search algorithm, interactive node replacement via clicking and dragging, live edge redrawing, and PNG export.

Across all three cases the GUI successfully: i) rendered correct edge mark semantics, ii) produced reproducible exports (PNG, edge list, comparison report), and iii) provided beginner friendly explanations for missing edges in the learned graphs. The comparison view highlighted algorithmic differences, helping relate algorithm choice to DAG structure. Overall, these examples demonstrate the reliable functionality of this tool across small, synthetic, and real-world datasets at interactive speeds providing a responsive user experience.

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 8.3.1: (A) PC DAG on LUCAS dataset (α=0.05, depth=-1). (B) GFCI DAG on Sachs dataset (α=0.05, depth=-1). (C) FGES DAG on ASIA dataset (penalty=2.0).*

*A screenshot of a computer

AI-generated content may be incorrect.*

*Figure 8.3.2: PC Search edge list on LUCAS dataset (α=0.05, depth=-1). Rows list pairs of variables with the specific edge mark returned by the search algorithm.*

*A screenshot of a medical report

AI-generated content may be incorrect.*

*Figure 8.3.3: (A) Orientation/type disagreements between PC, GFCI, and FGES run on the LUCAS dataset (α = 0.05, depth = −1; FGES penalty = 2.0). Rows list pairs of variables with the edge marks returned by each algorithm. (B) Unique edges per algorithm run on the same data (only GFCI produced unique edges with these settings). Rows list pairs of variables with the edge mark uniquely returned by the specific algorithm.*

*A screenshot of a cell phone

AI-generated content may be incorrect.*

*Figure 8.3.4: “Why not connected?” tab in the GUI for PC search on LUCAS (α = 0.05, depth = −1). Selecting “Born An Even Day” and “Allergy” detects the separating set S = {} (p = 0.1365) which indicates no evidence of dependence at the selected significance threshold.*

## 8.4 Comparison to Existing Tools

To evaluate this prototype, I compared it against current causal tools: TETRAD’s desktop GUI, GeNIe, and Dagitty. The comparison involved factors matching this project’s goals: i) usability and workflow, ii) explainability/readability of results, and iii) exports & reproducibility. The PC graphs on LUCAS (α=0.05, depth=–1) show complete agreement on edges across my GUI and TETRAD’s output, confirming correctness. However, Figure 8.4.1 highlights a usability difference. The default hierarchical layout my GUI uses avoids any edge overlaps whereas in TETRAD’s circular layout it shows “Born An Even Day” being connected due to overlapping the edge between “Attention Disorder” and “Car Accident”. This requires manual moving of the node to discover no relationship is present. This supports the choice to prioritise legibility by design. In short, the causal graphs match across tools, validating the correctness of my GUI with the primary differentiator of this project being usability. This encompasses simplified parameterisation, clearer layouts, and integrated explainability (algorithm comparisons and “Why not connected?”) with minimal setup.

Figure 8.4.2 (parameter dialogs) demonstrates the GUI’s trade-off between customisability and approachability. TETRAD contains a wider range of advanced options for the user to customise, whereas my dialogs only expose essential parameters for the chosen algorithm (α/depth/test type for PC & GFCI; penalty for FGES). These parameters all have plain language hints to accompany them. This reduces “parameter anxiety” for first time users with limited understanding of how these settings affect the output. Figure 8.4.3 contrasts TETRAD’s broader algorithm chooser with my tool list containing the three implemented search algorithms (PC, GFCI, FGES). This reflects the scope decision discussed in Chapter 4-5 opting for narrower algorithm coverage, but emphasis on guidance and simpler decisions.

Figure 8.4.4 shows the comparison table summarising differences between current causal discovery tools and my GUI. Dagitty (web application) is excellent for designing causal diagrams, d-separation and adjustment identification. However, it does not feature the ability to run searches on local CSV files and requires an internet connection to use. GeNIe’s purpose is focused on Bayesian networks and decision analysis. It can learn Bayesian network structures but is not intended for algorithm comparison or separating set explanations which my application focuses on. TETRAD is the most comprehensive causal discovery tool and is open source allowing for my GUI to call its algorithms. My GUI instead emphasises usability with tutorial wizards and tooltips, explainability with the “Why not connected?” tab, separating sets, edge mark meanings, and per-run comparison reports. It is also tailored for offline use with the ability to run searches on local CSV files and export outputs offline, just as TETRAD’s desktop GUI does.

In summary, the prototype GUI complements existing tools by lowering the barrier to entry for new students to try running PC/GFCI/FGES searches on their own observational data and aiding in interpreting results. The main limitations relative to TETRAD and GeNIe respectively are fewer algorithms/options which experts in the field would desire, and no time-series/intervention tooling. These are trade-offs acknowledged for designing a beginner-friendly GUI and are listed as future work in Chapter 10.

A diagram of a diagram

AI-generated content may be incorrect.

*Figure 8.4.1: PC Search on LUCAS dataset (α = 0.05, depth = –1). (A) my GUI’s hierarchal Graphviz layout containing no confusing overlaps vs (B) TETRAD’s desktop GUI’s circular layout, which can visually overlap (“Born An Even Day” overlapping neighbouring arrows) by default. Edge sets are identical between the two DAGs.*

*A screenshot of a computer

AI-generated content may be incorrect.*

*Figure 8.4.2: Parameter dialogs for PC search: (A) my GUI’s minimal format (title, α, depth, data type) targeted towards first-time users, and (B) TETRAD’s advanced dialog housing many additional options for expert tuning of the search parameters.*

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 8.4.3: Algorithm selection: (A) my GUI’s focused tool list (Import Data, PC Search, GFCI Search, FGES Search) to reduce choice overload for new users, and (B) TETRAD’s vast array of algorithms with broad coverage for expert users at the cost of higher complexity and a more overwhelming choice.*

A table with text on it

AI-generated content may be incorrect.



*Figure 8.4.4: Comparison of causal tools (My GUI, TETRAD’s desktop GUI, GeNIe, Dagitty web application). Categories compared: Discovery coverage, onboarding UX, explainability features, export options, offline / cross-platform support, licensing/cost, and notable features.*

## 8.5 Limitations & Threats to Validity

This section highlights validity threats to the claims made throughout Chapter 9 and addresses mitigations in place. The GUI demonstrates its goals were met (Chapters 4-8), but there are several limiting factors to generalisation.

**Algorithm coverage:** The tool integrates three causal discovery algorithms (PC, GFCI, FGES) on tabular, independent and identically distributed data. However, it does not cover time-series/temporal models, or non-linear methods (e.g. LiNGAM/NOTEARS), or mixed-type dataset modelling. **Impact:** findings cannot be generalised to those settings. **Mitigations:** the scope of this project is stated throughout the report. Future work discussed in Chapter 10 aims to broaden algorithm support and cover time-series data.

**Scale & layout:** Evaluation focused on graphs learned from small to moderate sized datasets (e.g. 8-12 variables, ≈2000 measurements). Very high dimensional datasets could cause issues with overly long runtimes and readability with overcrowded outputs. The hierarchical graph layout also depends on Graphviz (with a spring-layout fallback) and can still generate endpoint overlaps. **Mitigation:** manual click and dragging of variables, seeded layouts, and clear endpoint rendering are implemented. Future work will focus on zooming/panning, label overlap avoidance, and background algorithm execution.

**Explainability approximations:** “Why not connected?” uses a bounded search (looks at small candidate sets with capped conditioning sizes) and heuristic tests. These are dependent on the selected data type. The separating sets generated for explainability are only illustrative rather than guaranteed explanations of edge cancellation. **Mitigation:** the UI text describes results as explanations rather than proofs. Future work will expose exact sepsets found by the search algorithm run when available.

**Measurement validity:** Performance timings (Figure 8.4.1) were taken on a single MacBook with three test runs per action. UI latency < 300 ms was also untested for all UI interactions. **Impact:** limited external validity of recorded timings. **Mitigation:** hardware specifications and raw tests are mentioned in this report, and planned future testing involves cross-platform testing and UI event timing.

**User study limits.** The usability survey (Chapter 8.3) targets novice users and involves a survey with self-report scales. The sample size is small, and the order of tasks could threaten validity. Participants took the survey only after the GUI demo, which may introduce learning effects. **Impact:** conclusions may not generalise to expert users. **Mitigation:** future iterations of user testing will increase the participant pool size, randomise task order, and involve a more standardised measurement (e.g. SUS).

**Tool comparisons:** The side-by-side comparison with TETRAD’s desktop GUI (Chapter 9.4) used search algorithms and identical parameters on three different datasets. Differences in default parameters which are not exposed in my GUI can affect outputs. **Mitigation:** comparisons are to convey qualitative UX contrasts. The DAG comparison across the two tools uses edge list equivalence to verify my GUI’s correctness, rather than claiming superiority.

**Environment/version dependence & reproducibility:** My GUI’s functionality depends on py-tetrad, Java and Graphviz versions and platform. **Mitigation:** versions are pinned in the READ.ME document.

Taking these limitations into account, the evidence supports the project’s main aims: a beginner-friendly, student-oriented, offline GUI capable of learning correct causal graphs for supported algorithms. It provides lightweight explanations and is responsive on small-medium sized datasets (beneficial for teaching and benchmarking algorithms). The clear focus to strengthening the validity of this project is broadening the number of implemented algorithms, UI latency measurements, parallel algorithm execution, and a larger, more balanced user study.

# ***Chapter 9: Conclusion***

## 9.1 Summary of Contributions & Findings

This project delivers a beginner-focused, offline GUI for causal discovery, integrating PC, GFCI, and FGES via py-tetrad. It standardises edge-mark semantics (-->, ---, o->, o-o), and adds explainability using an Edges view, “Why not connected?” sepset tab, and Compare tab which can be exported. The controller provides robust fallbacks (Graphviz -> spring layout), caching to avoid redundant runs, and one-click exports (PNG, edge list TXT, comparison TXT). Unit and integration testing confirmed reliability (fallback, cache correctness) and correct end-to-end behaviour of the application. A task-based study with 7 novice students reported high ease of use (4.7-4.9/5) and moderate mental effort while using the GUI. Performance timings on a MacBook Air M2 showed responsiveness was well within the required timing windows. All must have functional requirements were met (with targeted should have a could have deferrals), and most non-functional requirements passed (one untested UI latency measure, and two non-critical should have requirements deferred). Side-by-side runs with TETRAD’s GUI produced matching edge sets showing correctness of this GUI, and the prototype improved default readability and beginner onboarding.

## 9.2 Practical Impact

The tool lowers the barrier to entry for students and novice researchers to run a standard set of causal searches on their own observational data. It enables them to interpret CPDAG/PAG outputs, and export reproducible graphs and reports without command-line skills or a background in computing. In teaching, it fits live demonstrations, labs, and coursework. In early-stage research, it enables rapid sanity checks and algorithm comparisons before conducting a deeper analysis. Users are aided in forming correct mental models of algorithm behaviour and assumptions due to clear parameters, tutorials, and readable layouts. In turn, this improves transparency and reproducibility in day-to-day practice when learning to understand causal discovery.

## 9.3 Future Work

Planned work includes broader algorithm coverage (e.g. LiNGAM/NOTEARS, time-series/temporal models, and mixed-type data), avoidance of label overlap, timestamped error log, first time run setup assistance, and depth of testing (UI latency measurements, larger and more diverse user studies, cross-machine performance testing). Explainability can be strengthened with exact sepsets when available, and parameter-sensitivity “what if” helpers.

## 9.4 Closing Remarks

The prototype GUI meets its aim: a usable, transparent, reproducible GUI that helps beginners learn by doing while running causal discovery algorithms. By focusing on scope and simplicity over expert configurability, the app provides a clear path to understanding causality for education purposes. With target next steps (expanded algorithms, UI polish, and broader evaluation) the tool can develop into a dependable teaching aid and lightweight research application for causal discovery.

# ***References***

1. Adel, T. and de Campos, C. (2017) “Learning Bayesian Networks with Incomplete Data by Augmentation,” *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1). Available at: <https://doi.org/10.1609/aaai.v31i1.10883>.
2. Agile Business Consortium (2025) *DSDM Agile Project Framework*. Available at: https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html (Accessed: August 26, 2025).
3. Amershi, S. *et al.* (2019) “Guidelines for Human-AI Interaction,” in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 1–13. Available at: <https://doi.org/10.1145/3290605.3300233>.
4. Andersson, S.A., Madigan, D. and Perlman, M.D. (1997) “A characterization of Markov equivalence classes for acyclic digraphs,” *The Annals of Statistics*, 25(2). Available at: <https://doi.org/10.1214/aos/1031833662>.
5. Bareinboim, E. and Pearl, J. (2016) “Causal inference and the data-fusion problem.,” *Proceedings of the National Academy of Sciences of the United States of America*, 113(27), pp. 7345–52. Available at: <https://doi.org/10.1073/pnas.1510507113>.
6. Carnegie Mellon University (2023) *py-tetrad*. Available at: https://github.com/cmu-phil/py-tetrad (Accessed: August 26, 2025).
7. Carnegie Mellon University (2025) *Tetrad: Education Materials - Documentation*. Available at: https://www.cmu.edu/dietrich/philosophy/tetrad/educational-materials/technical-details/documentation.html (Accessed: August 29, 2025).
8. Chickering, D.M. (2002) “Optimal Structure Identification With Greedy Search,” *Journal of Machine Learning Research 3*, pp. 507–554.
9. International Organization for Standardization (2020) *ISO 9241-110:2020 Ergonomics of human-system interaction — Part 110: Interaction principles*. Geneva. Available at: https://www.iso.org/standard/75258.html (Accessed: August 26, 2025).
10. *ISO/IEC 25010:2023 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model* (2023). Geneva. Available at: https://www.iso.org/standard/78176.html (Accessed: August 26, 2025).
11. Kalisch, M. and Bühlmann, P. (2005) *Estimating high-dimensional directed acyclic graphs with the PC-algorithm*.
12. Maathuis, M.H., Kalisch, M. and Bühlmann, P. (2009) “Estimating high-dimensional intervention effects from observational data,” *The Annals of Statistics*, 37(6A). Available at: <https://doi.org/10.1214/09-AOS685>.
13. Nielsen, J. (1992) “Finding usability problems through heuristic evaluation,” in *Proceedings of the SIGCHI conference on Human factors in computing systems  - CHI ’92*. New York, New York, USA: ACM Press, pp. 373–380. Available at: <https://doi.org/10.1145/142750.142834>.
14. Nielsen, J. (1994) *10 Usability Heuristics for User Interface Design*, *Nielsen Norman Group*. Available at: https://www.nngroup.com/articles/ten-usability-heuristics/ (Accessed: August 24, 2025).
15. Nogueira, A.R. *et al.* (2022) “Methods and tools for causal discovery and causal inference,” *WIREs Data Mining and Knowledge Discovery*, 12(2). Available at: <https://doi.org/10.1002/widm.1449>.
16. Ogarrio, J.M., Spirtes, P. and Ramsey, J. (2016) *A Hybrid Causal Search Algorithm for Latent Variable Models*.
17. Palmer, Z.B. and Oswal, S.K. (2025) “Constructing Websites with Generative AI Tools: The Accessibility of Their Workflows and Products for Users With Disabilities,” *Journal of Business and Technical Communication*, 39(1), pp. 93–114. Available at: <https://doi.org/10.1177/10506519241280644>.
18. Pearl, J. (2000) *CAUSALITY: MODELS, REASONING, AND INFERENCE*.
19. Pearl, J. (2009) *Causality*. Cambridge University Press. Available at: <https://doi.org/10.1017/CBO9780511803161>.
20. Peter Spirtes, Clark Glymour and Richard Scheines (2001) “Causation, Prediction, and Search,” in *Causation, Prediction, and Search*. The MIT Press, pp. i–x. Available at: <https://doi.org/10.7551/mitpress/1754.003.0022>.
21. Ramsey, J. *et al.* (2017) “A million variables and more: the Fast Greedy Equivalence Search algorithm for learning high-dimensional graphical causal models, with an application to functional magnetic resonance images.,” *International journal of data science and analytics*, 3(2), pp. 121–129. Available at: <https://doi.org/10.1007/s41060-016-0032-z>.
22. Ramsey, J.D. *et al.* (2018) “TETRAD—A toolbox for causal discovery,” in *Proceedings of the 8th International Workshop on Climate Informatics (CI 2018)*. Boulder, CO, USA: NCAR/UCAR.
23. Richardson, T. and Spirtes, P. (2002) “Ancestral graph Markov models,” *The Annals of Statistics*, 30(4). Available at: <https://doi.org/10.1214/aos/1031689015>.
24. Sachs, K. *et al.* (2005) “Causal Protein-Signaling Networks Derived from Multiparameter Single-Cell Data,” *Science*, 308(5721), pp. 523–529. Available at: <https://doi.org/10.1126/science.1105809>.
25. Schwarz, G. (1978) “Estimating the Dimension of a Model,” *The Annals of Statistics*, 6(2). Available at: https://doi.org/10.1214/aos/1176344136.
26. Wilson, G. *et al.* (2017) “Good enough practices in scientific computing,” *PLOS Computational Biology*, 13(6), p. e1005510. Available at: <https://doi.org/10.1371/journal.pcbi.1005510>.
27. Xun Zheng *et al.* (2018) “DAGs with NO TEARS: Continuous Optimization for Structure Learning,” *Proceedings of the 32nd International Conference on Neural Information Processing Systems* [Preprint].

# ***Appendix 1 – GitLab repository***

GitLab project repository link address - <https://git.cs.bham.ac.uk/projects-2024-25/jmk437>

**Contents:**

* **tetrad\_gui/ –** application source
* **requirements.txt –** Python dependencies
* **README.md –** information about the project
* **Project\_Definition.docx –** early project definition for project proposal
* **GUI\_Design\_prototypes.docx –** early mock GUI design prototype created with Figma
* **Project Demonstration.pptx –** PowerPoint slides for the project demonstration
* **.gitignore –** excluded local files when developing and testing the app
* **src/ –** includes main code (app.py, controllers.py, dialogs.py, tetrad\_iface.py, styles.qss, ui/main\_window.ui)

**how to run:**

1. Download the TetradStudentGUI-mac-arm64\_\_2\_.zip folder
2. Unzip the folder
3. Open the TetradStudentGUI.app file by double-clicking
4. If blocked by macOS Gatekeeper, right-click and press open, then open again (only necessary for first launch)
5. Double-click to start (runs fully offline)

The app has been tested on macOS 13.2 (MacBook M2, 8GB RAM). The app is designed to work offline, and only local files are accessed.

Troubleshooting:

* If Java/Tetrad is missing, install a JDK and relaunch (tested with Amazon Corretto 21 JDK)
* If Graphviz is not found, the layout falls back to spring layout, but installing Graphviz will use the neat hierarchical layout
* Invalid CSV shows an error dialog, and fixing headers/datatypes will solve the issue

# ***Appendix 2 – User Testing Survey***

Collected using Google Forms

















