School of Information Technology and
Engineering at the ADA University

School of Engineering and Applied Science
at the George Washington University

DESIGN AND IMPLEMENTATION OF A GRAPH ANALYTICS PACKAGE FOR MEDLEY INTERLISP

A Thesis
Presented to the Graduate Program of Computer Science and Data Analytics
of the School of Information Technology and Engineering
ADA University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science and Data Analytics
ADA University

By
Farid Aliyev

April, 2025

THESIS ACCEPTANCE

This Thesis by: Farid Aliyev
Entitled: *Design and Implementation of a Graph Analytics Package for Medley Interlisp*

has been approved as meeting the requirement for the Degree of Master of Science in Computer Science and Data Analytics of the School of Information Technology and Engineering, ADA University.

Approved:

Stephen Kaisler	22/08/2025
(Adviser)	(Date)
Abzatdin Adamov	24/08/2025
(Program Director)	(Date)
Abzatdin Adamov	24/08/2025
(Dean)	(Date)

ACADEMIC INTEGRITY STATEMENT

“I affirm that this is my own work, I attributed where I used the work of others, I did not facilitate academic dishonesty for myself or others, and I used only authorized resources for my Thesis, per the ADA University Academic Integrity requirements. If I failed to comply with this statement, I understand consequences will follow my actions. Consequences may range from failing the course to expulsion from the program/university and may include a transcript notation.”

Farid Aliyev		27/08/2025
(Full Name)	(Signature)	(Date: DD.MM.YY)

ABSTRACT

Graph analytics is a critical field today for understanding the organization and behavior of intricate systems, and its applications are ubiquitous in social networks, transportation systems, scientific computing, and artificial intelligence. While modern programming environments such as Python and R provide extensive libraries for graph analytics, legacy environments such as INTERLISP lack specialized and structured tools to deal with such analyses. This thesis addresses this gap by creating a comprehensive, modular graph analytics package for INTERLISP, founded on its object-oriented extension LOOPS.

The primary focus of the work was to produce a fully functional, reusable, and extensible framework that performs basic graph operations, traversals, and analytical computations in the boundaries and symbolic processing approach of INTERLISP. It has structural graph operations (manipulation of vertices and edges), standard graph traversals (breadth-first search), finding shortest paths, centrality measurements (betweenness, closeness, eigenvector), cluster analysis, tests of connectivity, and subgraph identification. All the functionality was coded to operate within the INTERLISP system without calling upon external tools, thus maintaining the self-contained nature of the system.

Particular emphasis was laid on usability, modularity, and computation transparency. Both manual and file-based graph construction was supported so that users could carefully build and experiment with graph structures. Human-readable reports of graph-level and node-level analysis were automatically generated, along with a debug mode for step-by-step tracing of algorithms. The integration of Common Lisp features with INTERLISP structure guaranteed smooth integration with the Medley system environment.

The system was thoroughly tested with a variety of graphs ranging from small manually created instances to relatively medium-sized ones with dozens of vertices and edges. Results proved the correctness of all the algorithms run, scalability of the design, and the feasibility in general of running high-end graph analytics on a legacy symbolic programming framework. In addition, problems unique to INTERLISP, including implementation of basic data structures as well as the adaptation of algorithms for symbolic processing, were overcome.

This work is the first recorded attempt to develop a specialized graph analytics package in INTERLISP LOOPS and demonstrates that it is feasible for legacy systems to be made to handle modern-day computational needs when approached with the appropriate design principles. The package has not only immediate utility for researchers and developers still working with INTERLISP but also provides a foundation for additional improvements, including directed graph support and graphical visualization. Also, the project serves as a general illustration of how older computing environments can be revitalized through careful engineering so that they will continue to be useful in the modern computational era.

Keywords: INTERLISP, LOOPS, graph analytics, symbolic processing, legacy systems, object-oriented Lisp, centrality, graph algorithms, Medley environment, retrocomputing

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Explanation
Lisp	LISt Processing
LOOPS	Lisp Object-Oriented Programming System
BFS	Breadth-First Search
GNNs	Graph Neural Networks
BSMS-GNN	Bi-Stride Multi-Scale Graph Neural Network
DWIM	Do What I Mean
IL	INTERLISP
CL	Common Lisp

1 INTRODUCTION

Graph, network, and mesh analytics are essential tools in contemporary computing, which find application in the analysis of social networks, transportation modeling, scientific computing, and artificial intelligence. These techniques make it possible to model and investigate structured relationships between entities and thus play an important role in both theoretical and applied applications. Modern programming languages like Python, R, and Java feature extensive frameworks that support graph analytics, but such functionality is absent in the INTERLISP environment. This work, therefore, addresses the development of such an analytics package for INTERLISP and enables the modernized system to perform computations on graph-based structures.

While INTERLISP was once extremely significant in AI research, employing it today entails practical difficulties. The language's unique and often extremely dense syntax, combined with the absence of current documentation or community resources, makes it hard for new programmers to learn or use. This research project is one of the few modern efforts to extend the capabilities of INTERLISP, not only by adding new capabilities, but by providing a hierarchical and functional codebase that can serve as an example and a development tool for others. The project thus contributes to the revitalization of a legacy language both through technical development and practical accessibility.

As aforementioned, INTERLISP is a powerful interactive environment based on Lisp that played a significant role in the work of artificial intelligence efforts during the 1970s and 1980s. An important extension is LOOPS, or the Lisp Object Oriented Programming System to INTERLISP, a more convenient representation of complicated data structures in an object-oriented manner. Using these capabilities, this research aims to design and develop an object-oriented analytics package that handles the following:

- Graphs: The basic node-edge structures used in traditional graph theory.
- Networks: Graph structures where edges may have weights or multiple labels.
- Meshes: Multi-dimensional structures that contain hierarchical and hypergraph relationships.

Despite its modernized capabilities, the lack of an integrated framework in INTERLISP LOOPS for graph and network analytics makes it difficult for researchers and developers to perform complex data analysis within this environment. Many real-world problems extend beyond basic graphs, requiring network-based and mesh-based representations to capture the relationships more accurately. For these reasons, algorithms designed for traditional graphs must be modified to function within an object-oriented symbolic processing system like INTERLISP LOOPS.

The proposed package is intended to be helpful to a wide category of users who are dealing with INTERLISP, from researchers keeping older systems alive, to programmers creating new tools in INTERLISP, and learners attempting to understand the structure and potential of the language. By offering a well-documented, modular, and structured approach to implementing a non-trivial analytics system in INTERLISP LOOPS, the project sets a precedent that can be emulated by others—either for similar analytic systems or for application of different extensions. This contributes not only to the area of graph analytics but also to the greater goal of making legacy environments useful and relevant today.

Moreover, unlike modern packages such as *NetworkX* of Python or *igraph* of R, which have built-in tools for the validation and visualization of graphs, INTERLISP does not have any internal graph operation validation facility. This calls for a self-contained approach where correctness is ensured entirely within INTERLISP, independent of external validation tools.

One of the main goals of this research is understanding how INTERLISP LOOPS can be applied in an effective way for graph analytics. In contrast to a common language that would have a rigid graph processing model, INTERLISP LOOPS's symbolic processing model and object-oriented nature necessitate a different kind of approach. Such capabilities are to be explored for the possible designing and implementation of algorithms solving graph traversals, shortest path computation, connectivity analysis, and structural measures in an efficient manner. Full exploitation of the class-based structure and dynamic method binding of LOOPS can yield a modular, extensible analytics package that fits well into the INTERLISP environment and is computationally feasible.

A second key question is how general graph algorithms must be altered to perform optimally in more constrained and richer environments. General algorithms are framed in terms of basic, homogeneous graph structures and don't deal with special relationships or constraints nodes and edges might have. In more sophisticated systems, algorithms may need to incorporate custom logic to handle personalized traversal rules, prefer some connections over others, or enforce structural integrity. This study aims to develop a framework inside INTERLISP LOOPS that extends conventional algorithms to include such complications, and that does so while being both correct and efficient.

This research is also grounded in classical graph theory, where structures like nodes and edges, traversals, shortest paths, and connectivity measures remain central to computational modeling. The realization of these well-established algorithms to run in a symbolic and object-oriented setting like INTERLISP LOOPS demonstrates the evolving relationship between theoretical foundations and implementation environments. By re-engineering traditional algorithms in a limited legacy environment, this project demonstrates that basic principles of graph theory remain highly portable, and that with thoughtful design, they can be executed efficiently even outside of modern environments.

Lastly, validation of the analytics package implemented has to be within INTERLISP. Although existing graph processing packages allow for the use of checking results against a set of precomputed results to ensure that all is well as it should be, no such checks are feasible in INTERLISP. Internal consistency checks need to be implemented to check for correctness, as well as controlled test environments and algorithmic steps for correctness. This will ensure that the package developed doesn't depend on any other external tools and executes completely within the INTERLISP environment.

The research aims to make a significant contribution to graph analytics and INTERLISP LOOPS modernization through the creation of a new object-oriented package for analytics in this context. Analysis of graph and network structures currently not supported by tools will become feasible through the package. The research also extends most popular algorithms with the intention of making the algorithms operate on complex structures within constraint environments and also leveraging the object-oriented properties of INTERLISP LOOPS.

The second significant contribution is that of having an in-built validation environment in INTERLISP LOOPS without any third-party software. Only through having internal verification facilities to verify the correctness of each algorithm using well-structured test cases and consistency checks can INTERLISP ensure correctness. That way, INTERLISP will be much more capable of doing graph analytics reliably without needing any external software verification.

Aside from its immediate technical impact, the project also shows the way in which contemporary analysis techniques can be applied to a legacy environment. Leaning on the capabilities of INTERLISP LOOPS, the project will illustrate how computational needs today can be made available by expanding systems like those. Techniques developed here would be able to serve as a basis of further investigation in legacy system modernization and illustrate how the earlier environments can still be used to address challenging computational problems today.

This thesis begins with the background and motivation, explaining in great detail INTERLISP and LOOPS, their importance, and preceding research on graph and mesh analysis. The methodology explains the package design and implementation, verification, chosen algorithms, and the way they are adapted. It is then succeeded by the results and appraisal part that contrasts the algorithm performance, efficiency, and scalability. Finally, it gives a preview of the contributions made and gives hints regarding how it can be enhanced in the future.

To give some background, INTERLISP was developed at Xerox PARC in the 1970s for doing AI research and symbolic computation. It had new features like built-in debugging, automatic memory management, and strong list-processing facilities that made it very suitable to deal with structured data. Despite these relative strengths, however, INTERLISP was still lacking any formal notion of object-oriented data manipulation and representation that was becoming progressively necessary in advanced computation. LOOPS (Lisp Object-Oriented Programming System) was created to extend INTERLISP with object-oriented programming features such as classes and instances to hold data in a structured format. It provided support for inheritance, dynamic method binding, which provided immense

flexibility in reusing algorithms for varying data structures. Its hierarchical representation of data also made it particularly well-suited for graph and network representation.

One of the greatest challenges to building this package is the absence of utility functions and supporting libraries as common to today's environments. INTERLISP, unlike R or Python, requires even the most routine data-handling machinery and forms to be built manually, fitted to the quirks of Lisp's symbolic representation. This restriction necessitates a bottom-up engineering style that, although difficult, offers full control of system design and algorithmic structure. Second, the project is inspired by today's graph analysis libraries but inherits from and builds upon only their concepts in a form suitable for the LOOPS environment. The focus has been on simplicity and usability so that users can meaningfully interact with the system regardless of the limitations of the base language.

Most of the existing work in the domain of graph analytics is concentrated on modern programming environments, where different libraries have implemented extensive amounts of built-in functions for traversal, computation of the shortest paths, centrality calculations, and many other operations related to graphs. However, these frameworks are designed for either procedural or object-oriented languages and are thus not suited for symbolic processing environments like INTERLISP.

In conclusion, this project satisfies a challenging task: not only does it fill a long-standing gap in the capabilities of INTERLISP, but also demonstrates the degree to which modern analytic insight can be embedded within a legacy programming model. By developing a user-friendly, object-oriented graph analysis package for INTERLISP LOOPS, the project demonstrates the adaptability of classical algorithms to constrained symbolic environments. The work accomplished here goes beyond the technical implementation level, offering not only a model for future tool implementation within INTERLISP, but also a case study in the modernization of legacy systems through selective design and adaptation. The following chapters offer a more extended treatment of the historical and technical context of the work, the methodological framework of implementation, the outcomes of system evaluation, and implications for research and development in the future.

2 LITERATURE REVIEW

Lisp, an acronym for "LISt Processing", is a high-level programming language one of the earliest, designed in the late 1950s by John McCarthy. Its development was revolutionary and had a completely parenthesized prefix notation and homoiconic (code acting upon data) features, which well suited it as a tool in artificial intelligence programming and symbolic computing.

Interlisp first appeared in the early 1970s as a significant Lisp dialect, which was developed at Xerox PARC. Interlisp was initially developed as an integrated programming environment, which was a programming language combined with strong software development tools like powerful editing and debugging capabilities. Interlisp's interactive development emphasis and full environment made it an excellent place for AI research, computational linguistics, and graphical user interface research.

The most notable extension of Interlisp is the Lisp Object-Oriented Programming System (LOOPS), which brought object-oriented design to a legacy language setting [1]. LOOPS brought the feature of defining and manipulating objects, hence allowing the programmer to effectively model complex data structures. The extension was critical in facilitating AI research and complex system modeling in the Interlisp setting.

Graph analysis is a very vibrant field that started in conventional graph theory and has big-scale modern applications in networks such as transportation and social networks all the way to scientific computing. There is sufficient material in the current literature to be applied in the formulation of the fundamental algorithms as well as the modern frameworks that implement these methods.

Early graph theory laid many of the foundations that are used today. For example, Dijkstra's shortest path algorithm [2] and Bron–Kerbosch algorithm for listing all cliques of an undirected graph [3] are still landmark papers. Further, centrality measures—betweenness, closeness, and eigenvector centrality—were developed to quantify the influence of nodes in networks. These have been complemented with different variants such as power centrality and Freeman degree centrality, which contribute further to our understanding of the network structures and robustness [4].

Modern graph processing libraries as Python, Java, and R possess strong built-in functionality. For instance, NetworkX and others offer efficient data structures, visualization, and validation functionality that can support advanced methods—such as machine learning algorithms such as graph convolutional networks [5] [6]. The platforms are designed to handle the computational demands of massive networks and changing data.

The fusion of machine learning and graph theory has also advanced the field, for instance, in the application of methods such as graph neural networks and the MeshGraphNets model. The application integrates classic graph algorithms and deep learning to improve prediction for sophisticated domains such as structural mechanics and aerodynamics [7]. Mesh-based simulation, for instance, is central to modeling complicated physical systems across most fields in science and engineering. Mesh representations are susceptible to aggressive numerical integration methods and the degree of granularity in their representation can be altered to realize optimal trade-offs between accuracy and efficiency. High-dimensional scientific simulations are computationally expensive, however, and generally require solver and parameter tuning per system on a case-by-case basis. MeshGraphNets responds to this by the utilization of graph neural networks (GNNs) for the purpose of learning mesh-based simulations, i.e., predicting dynamics within various physical systems such as aerodynamics and structural mechanics. It is more efficient, with the simulations taking several times faster than the conventional methods.

Recent progress in graph analytics has largely been fueled through the addition of machine learning techniques, specifically Graph Neural Networks (GNNs). GNNs have performed remarkably well for learning and predicting physical systems described by mesh representations. As mentioned earlier, the MeshGraphNets methodology applies GNNs towards learning mesh-based simulations, which encompass dynamics of numerous physical systems like aerodynamics and structural mechanics. Not only does the approach improve accuracy, but it is also computation-cost-effective since simulations are completed quicker than conventional practice [7].

Another notable development is the Bi-Stride Multi-Scale Graph Neural Network (BSMS-GNN) which addresses challenges in modeling long-range interactions across unstructured meshes. By introducing a robust pooling strategy, BSMS-GNN improves the efficiency and accuracy of simulations, particularly in large-scale mesh-based physics applications [8]. All of these innovations highlight the evolving nature of graph analytics, driven by the intersection of traditional computational methods with modern machine learning techniques.

Despite the innovations seen in modern environments, legacy systems like INTERLISP have not been greatly impacted by these innovations. Some of the early documentation, as well as subsequent manuals on MEDLEY-LOOPS highlights INTERLISP's pioneering status in early AI research [1] [9] [10]. However, these sources also reveal limitations in terms of integrated graph processing capabilities. The lack of built-in libraries in INTERLISP to support graph validation, traversal, and other graph-based operations means that such tasks must be addressed with custom solutions. This deficiency is reinforced by research into legacy system modernization, where emphasis is placed on the difficulties in mapping contemporary computational paradigms to historic platforms [11].

The application of object-oriented programming paradigms has had a profound impact on the implementation and design of graph processing systems. OOP facilitates encapsulation of graph entities as classes and objects, thereby enhancing modularity and reusability of code. For INTERLISP's LOOPS, application of an object-oriented framework facilitated embedding graph processing functionality. Programmers are able to design classes for graph edges and nodes and include methods for operations such as traversal, searching, and manipulation. This made it easier to write more complex programs.

In contrast, other OOP paradigms have also been adapted to include graph processing. Java and Python, for example, have extensive libraries and frameworks, i.e., JGraphT and NetworkX, respectively, that include useful graph manipulation and analysis tools. The libraries gain strength from the attributes of OOP, enabling the creation of extensible and maintainable graph-based programs [5].

The object-oriented paradigm in graph processing makes development easier as well as application scalability and flexibility in a broad array of fields.

In addition, the symbolic processing capabilities provided by INTERLISP play an important role in supporting implementations of graph and network analysis. Dynamic data support and automatic memory management capabilities of the environment allow it to be able to process acyclic, evolving graph structures efficiently, which is very important in traversal and pathfinding algorithms. Moreover, features such as DWIM (Do What I Mean) and history tracing support iterative development and debugging in order to support rapid testing and optimization [1]. INTERLISP's symbolic capabilities combined with the object-oriented methodologies of LOOPS provide a solid basis for building an extensible package for analytics for the management of multi-labeled edges and hierarchical mesh connectivity, the subject of this research.

MEDLEY-LOOPS: The Basic System manual explains the object-oriented programming (OOP) features of LOOPS, a mandatory extension to INTERLISP that permits class hierarchies, inheritance, and dynamic method binding [9]. The facilities of OOP enable symbolic definition of graph entities like nodes, edges, and hyperedges, crucial to modeling complex relationships between graphs and networks. LOOPS' ability to offer modular and reusable code makes it an apt choice for use in scalable analytics package development, and therefore a good place for the research's goal of creating a full graph analytics toolkit.

MEDLEY-LOOPS: Tools and Utilities offers primitive development tools that facilitate the use and experimentation with graph analytics algorithms in INTERLISP LOOPS [10]. It includes interactive debuggers, dynamic class manipulation, and data inspection utilities, which are instrumental in error detection and correction in algorithm development. The utilities enable rapid prototyping and real-time alteration of graph representations, thus enabling algorithms to be developed efficiently.

The Neo4j Graph Algorithms covers a broad set of modern graph analytics methods, from shortest path calculations, community detection, and centrality analysis [13]. The algorithms are designed to maximize high-performance analytics for native graph databases' real-time usage. Algorithmic approaches and optimization methods in this book, e.g., the application of efficient traversal and memory management, are handy. These strategies can be brought forward to INTERLISP's symbolic processing paradigm, providing a modern perspective on how to embed strong graph analytics within an older system like INTERLISP LOOPS.

The literature considered here depict the significant gap: while modern programming environments accommodate expressive graph analytics toolkits, systems like INTERLISP lack such inherent ability. The gap here offers a rationale for the present research, which seeks to close the gap between classical algorithmic theory and modern analytical needs by re-engineering graph algorithms for INTERLISP LOOPS. Along the way, the research contributes to the revival of a historically important language as well as to the advancement of graph processing methods in the context of resource-constrained environments.

3 RESEARCH METHODOLOGY

The methodology employed in this research is based on a design- and implementation-oriented approach, appropriate to the special requirements of designing a graph analysis package within the environment of the INTERLISP LOOPS. Contrary to conventional experimental or data-oriented models, this effort focuses on the design of a software program that expands the functional capability of a legacy system through careful design and implementation of fundamental algorithms and data structures.

The research took a pragmatic, iterative development cycle, beginning with the development of fundamental functionality such as graph construction, traversal, and pathfinding. Once a core set of pieces was functional, additional layers of capability were added incrementally, tested, and iterated on. This allowed for continuous system evaluation and tuning, especially as more advanced graph operations and edge cases arose.

Because INTERLISP doesn't have any pre-written libraries or frameworks for graph analysis, all had to be implemented from scratch. This entails representing graphs in objects and classes in LOOPS, installing traversal and pathing algorithms, and creating a lightweight validation strategy that does it all within the environment itself. No external computational libraries or visualizing tools were used, making the point of needing an internally verifiable and self-contained system.

Throughout the design process, a set of core design principles guided decision-making:

- **Simplicity**: The inner operations and organization were made comprehensible and supportable, with no complexity beyond what is strictly needed.
- **Usability**: The organization of package functions and the interface are designed to provide an easy experience for users, even those with less INTERLISP experience.
- **Modularity and Reusability**: The core functions are designed with reusability in mind. The higher-level analytics methods are typically constructed by combining the core operations such that clean abstraction and future extension are facilitated.
- **Clarity**: Naming conventions, pattern-of-structure, and comments were all of high importance to make the codebase readable as a reference for others within the same context.

This approach also reflects the broader goals of the research: not only to build a technically capable package but also to demonstrate how legacy symbolic environments can be extended to meet modern analytical needs through well-structured, object-oriented design. The sections that follow explain in detail the development environment, system design, algorithm implementation, and validation methods that comprise the full methodological framework of the project.

3.1 Environment and Tools

The graph analytics library was created within the Medley INTERLISP platform, which runs perfectly on Windows and macOS systems. Even though the primary development platform was Windows, experimentation and testing were done on macOS as well to verify cross-platform functionality and performance. Medley offers a modernized implementation of INTERLISP, supplemented with the LOOPS (Lisp Object-Oriented Programming System) extension required for constructing advanced data structures and object-oriented programming techniques such as class definition and method dispatch.

The Medley INTERLISP environment is a bridge between new and old computing and accommodates the use of older symbolic programming facilities together with more modern language features. Symbolic processing facilities in INTERLISP are essential to graph data manipulation being dynamic, especially for traversal, pathfinding, and connectivity analysis types of operations. LOOPS builds on this support by adding object-oriented facilities, which allows code to be more modular and reusable. By combining these two systems, the development environment also offers support for both symbolic computation and interactive development simultaneously in a manner that is very well adapted to graph analytics.

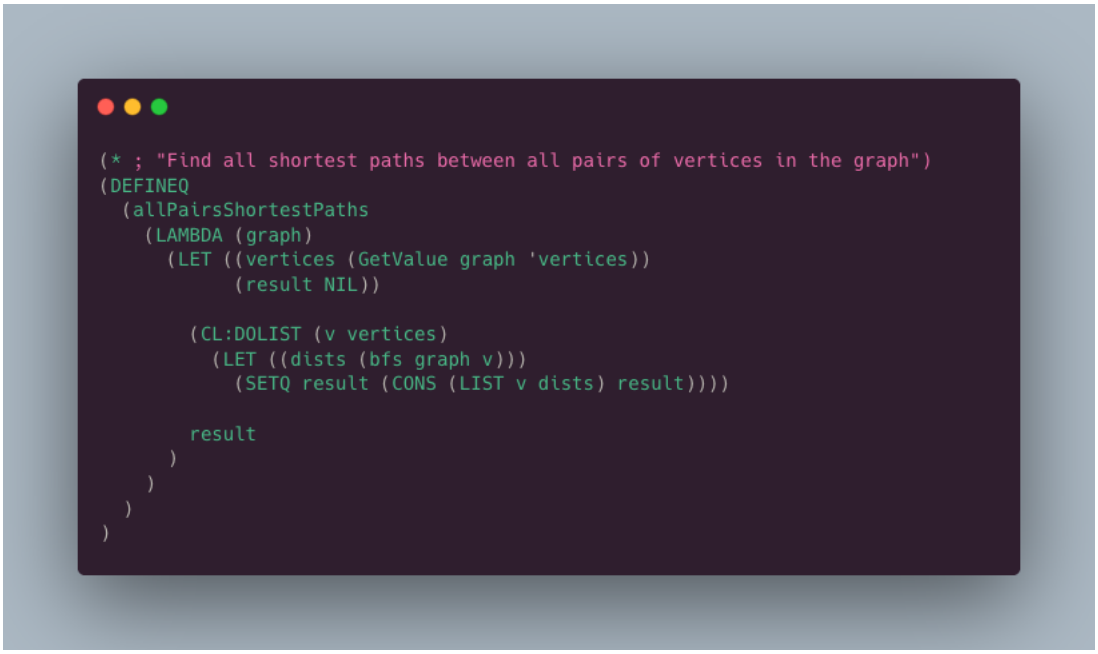
3.1.1 Integration of INTERLISP and Common Lisp

One of the strongest features of the Medley INTERLISP system is the ability to combine INTERLISP and Common Lisp and make it possible to use both programming languages during the same session of development. Common Lisp, being a newer version of Lisp, offers cleaner syntax, standard conventions, and more utilities than INTERLISP. This hybrid setting played an important role in facilitating the process of development by enabling the flexibility of using the facilities of Common Lisp while accommodating the symbolic processing facilities of INTERLISP.

In Medley, the interaction between the two dialects is facilitated by the fact that Interlisp and Common Lisp are integrated through their execution environment and tools, hence the use of the *IL:* and *CL:* prefixes (*IL* for Interlisp and *CL* for Common Lisp, respectively) invoke the right mechanisms. This allowed function calls, exchange of data, and object operations between the two dialects. For instance, Common Lisp's sophisticated list processing facilities were used to ease some tasks, i.e., graph structure parsing and data transformation. On the other hand, the symbolic environment of INTERLISP

was used for those activities that dealt with greater dynamic creation and manipulation of objects, i.e., graph data structure management and implementation of LOOPS-based object hierarchies.

This cross-language compatibility was beneficial, since programmers could capitalize on the advantages of each dialect for different purposes. When Common Lisp's syntax offered more readability and efficiency, INTERLISP's advantages in symbolic processing and dynamic object manipulation allowed the realization of graph structures (nodes, edges, and graphs) in an object-oriented, intuitive manner. The Medley system ensures clear integration, whereby Common Lisp functions can be accessed immediately from INTERLISP, and vice versa, without requiring difficult inter-process communication or outside bridging.



```
(* ; "Find all shortest paths between all pairs of vertices in the graph")
(DEFINEQ
  (allPairsShortestPaths
    (LAMBDA (graph)
      (LET ((vertices (GetValue graph 'vertices))
            (result NIL))

        (CL:DOLIST (v vertices)
          (LET ((dists (bfs graph v)))
            (SETQ result (CONS (LIST v dists) result))))

        result
      )
    )
  )
)
```

Figure 3.1: Code Snippet Demonstrating INTERLISP-Common Lisp Interoperability

Figure 3.1 demonstrates a code snippet that shows the interoperability between INTERLISP and Common Lisp within the Medley environment. The *allPairsShortestPaths* function was defined using the INTERLISP's *DEFINEQ* syntax construct but internally, it invokes the *DOLIST* macro from the Common Lisp namespace, as indicated by the *CL:DOLIST* prefix. This syntax tells us that the *DOLIST* construct is being used from the Common Lisp environment specifically, allowing the programmer to use its more advanced iteration semantics within still being inside INTERLISP. Cross-dialect use of this sort allows the blending of old symbolic-style constructs with newer programmatically structured Common Lisp functionality, allowing for flexibility in programming and enhancing expressiveness of the graph analytics package.

3.1.2 Platform and Development Environment

The primary development environment for this project was on a Windows platform, which provided a stable and compatible environment to run Medley. Although the system is considered legacy software, Medley was configured to run smoothly on Windows using an emulator or compatibility layer, enabling full functionality throughout the development and testing phases.

In addition, macOS was used for cross-platform testing. Being able to run Medley under both Windows and macOS ensured that the resulting package would be platform-agnostic to users who could potentially use the tool on alternate systems. Testing on macOS allowed for some of the platform issues to be found, although actual development work tended to be identical on both platforms.

INTERLISP's symbolic processing capabilities were crucial to the project. INTERLISP has the native capability to use lists and symbols, which allows dynamic manipulation of data during the

process of graph analysis. This feature of INTERLISP makes it well-suited to pathfinding and graph traversal algorithms, the latter of which requires dynamic real-time modification of data structures. The memory management and garbage collection features of Medley made it possible to manage memory effectively despite handling large graphs through the development and testing processes.

Medley's object-oriented extension, which was known as LOOPS, supported the specification of classes (i.e., for vertices, edges, and graphs) and dynamic method dispatch support. LOOPS supported modularization of graph components and reusable objects' setup for different graph operations. This modularity and reuse of objects were crucial in scaling the system and in ensuring that the package being built could be extended in the future, which would allow one to introduce new graph algorithms or functionality without compromising the existing codebase.

LOOPS' backing for class hierarchies, inheritance, and dynamic method dispatching also facilitated flexible extensibility of graph algorithms. For example, complex graph structures like multi-edges or weighted edges may be defined as subclasses of a general edge class, and graph processing procedures may be readily modified and extended.

No external tools were used, which required a reliance on manual documentation within the system and careful structural organization of the project files. However, clear file/module organization (e.g., *vertex.lsp*, *edge.lsp*, *graph.lsp*) ensured that different aspects of the graph analytics package were isolated and could be worked on independently.

In summary, this graph analysis package was developed entirely within the Medley INTERLISP environment and cross-platform tested on Windows and macOS. The combination of INTERLISP's symbolic processing model and LOOPS' object-oriented features allowed the building of an extendable, modular package that is capable of complex graph operations. The intermixing of Common Lisp with INTERLISP added further depth to the development cycle, being beneficial by using the strengths of both the dialects while maintaining compatibility throughout the system. Despite all the restrictions of using a legacy system, the development platform was robust and efficient for that specific purpose.

3.2 System Design and Architecture

The architecture of the developed graph analytics package illustrates a methodical and module-based approach towards developing an actual software system inside an older symbolical framework. Since there is no native support for graph analytics in INTERLISP, and modern libraries cannot be imported to the target framework, the complete system was designed from scratch utilizing object-oriented programming techniques provided by LOOPS. This section describes the class structure, component responsibilities, interconnections, and control flow that constitute the core of the package.

3.2.1 Overall System Architecture

The package is organized into properly isolated modules corresponding closely to core graph theory concepts. Each module corresponds to a central component of a graph structure:

- **Graph.lsp** – declares the top-level Graph class and procedures for the general control of the graph.
- **Vertex.lsp** – declares the Vertex class and procedures for node properties and connectivity.
- **Edge.lsp** – declares the Edge class and is responsible for between-vertex relationships.

Each of these modules has a set of responsibilities and operates independently but in coordination with the others. The Graph class acts as the central coordinator, maintaining lists of all vertices and edges. This offers centralized control and is a single point of entry for most analysis operations.

The architecture has a composition-based architecture, with the Graph class containing multiple Vertex and Edge objects and each Edge referencing the Vertex instances to which it connects. The structure supports bi-directional navigation of graph structures and recursive or iterative operations including traversal, pathfinding, and connectivity checks.

3.2.2 Class Responsibilities and Structure

- **Vertex Class**

The Vertex class is meant to represent a single node of a graph. Each Vertex object has:

- A unique *id* to distinguish it from the others.
- A *label* to store semantic information or identifiers.
- A list called *connectedVertices*, which stores direct references to other Vertex objects.

This last property is employed as an adjacency list so that efficient retrieval of neighbors is possible during traversal. The use of direct object references here avoids symbol table lookup or list scans and meet the performance constraints of INTERLISP.

- **Edge Class**

The Edge class is used to represent a connection between two vertices. It contains:

- A *from* field with a reference to the starting Vertex.
- A *to* field with a reference to the ending Vertex.
- A *label*, optional to mark the edge with semantic information.

Edges are stored within the Graph object as well as indirectly reflected in the *connectedVertices* list of every Vertex. This dual representation enables both edge-based and vertex-based algorithms to be implemented efficiently.

- **Graph Class**

The Graph class is a representation of the whole graph structure. It encompasses:

- A list of all the vertices in the graph.
- A list of all the edges.

All structural modifications are done through the Graph object to ensure consistency and integrity across related objects.

3.2.3 Object Relationships and Interaction

The design is strongly object-oriented, founded on explicit references between components to convey relationships. It is a reflection of classical object modeling practices used in existing OOP languages but in LOOPS realized through class definitions, inheritance (where required), and method calls.

- Vertices reference other vertices via *connectedVertices*, building a local representation of the graph structure.
- Edges have back-references to endpoints, allowing algorithms to identify source and destination nodes without traversal.
- Graph is the container and controller, providing global consistency and dispatching analytic operations.

Each class instance is employed as a first-class object with encapsulated data and behavior. Components either communicate by method call (e.g., calling a method on a Vertex to get its neighbors) or by directly manipulating object slots with LOOPS mechanisms.

Figure 3.2 illustrates the object-oriented structure of the package, showing how the Graph, Vertex, and Edge classes are related. The Graph class holds both vertices and edges, and Edge objects maintain directional references back to their related Vertex instances. Vertex objects maintain local adjacency information in a list of adjacent vertices.

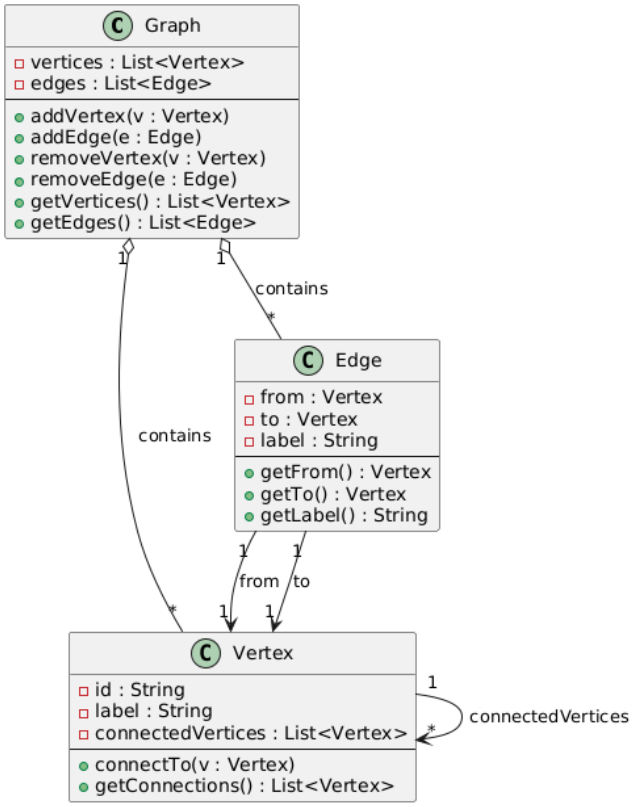


Figure 3.2: UML Class Diagram of the Graph Analytics Package

3.2.4 Design Principles

The system's architecture was influenced by a collection of interdependent design principles:

- **Modularity**: Code is structured into modules along lines of logical function, enabling maintenance and expansion.
- **Reusability**: Lower-level helper functions (e.g., accessing neighbors, connectivity tests) are reused by higher-level methods.
- **Simplicity**: The system avoids overcomplication and favors direct data flow using simple list structures and minimal indirection.
- **Clarity**: Functions, methods, and variables have descriptive names to enable readability and maintainability.
- **Extensibility**: The design is made in a way that it is possible to add new algorithms (e.g., clustering, centrality) with minimal disruption to existing code.

This controlled approach not only makes the package work as it does now but also make it ready for extension in the future.

3.2.5 Data Flow and Execution Model

Data flows through the system in an object-oriented and event-driven fashion:

1. The user creates an instance of Graph and populates it with Vertex and Edge instances.
2. Upon request for an operation (e.g., traversal), the Graph instance invokes internal functions which recursively send messages to Vertex and Edge instances.
3. Results (e.g., paths, nodes visited, boolean flags) are returned to the caller as functional output or change in graph state.

There is no use of global symbol tables or external data stores. All computation is object-local and in-memory, as is consistent with the symbolic nature of Lisp.

Users can dynamically create graphs, invoke analytics operations, and inspect or modify graph state at runtime. This dynamic and interactive model maps well to INTERLISP's past history of live debugging and exploratory programming.

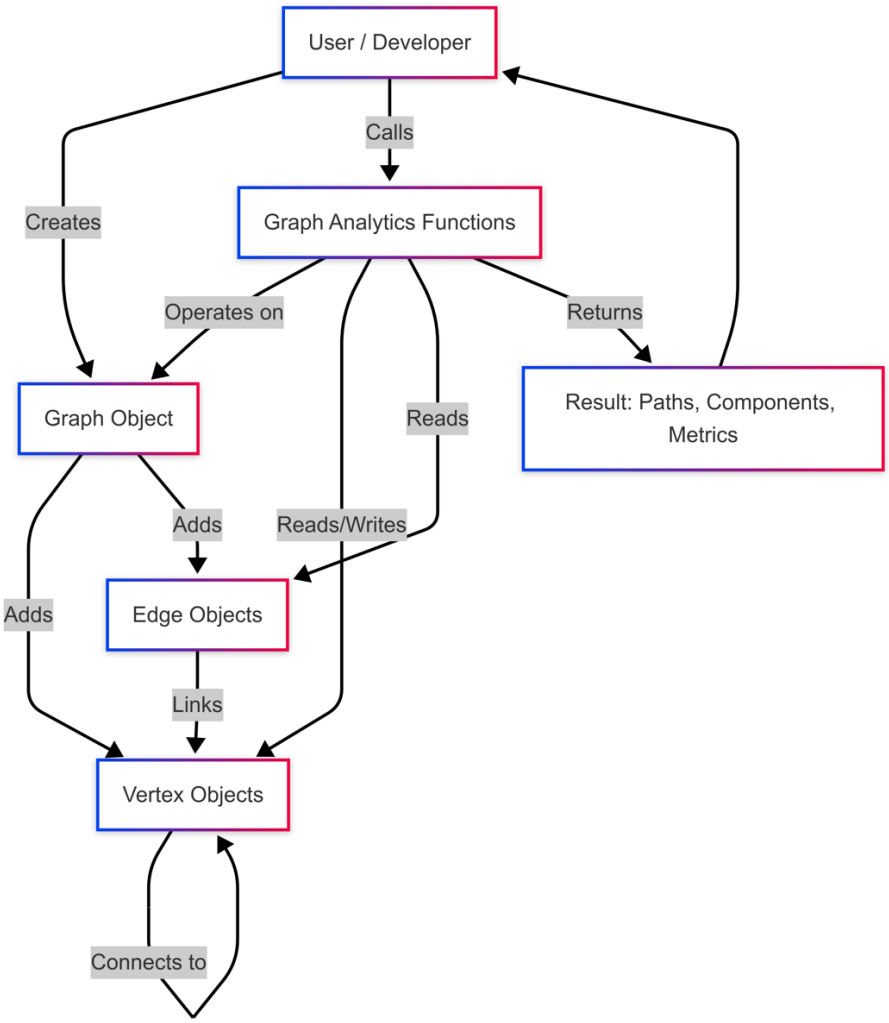


Figure 3.3: Data Flow Diagram of the Graph Analytics Package

Figure 3.3 illustrates the data flow within the system from user-created graph construction to analytics function execution. Graph objects control operations between vertex and edge instances, and results are reported back to the user based on algorithmic processing.

Lastly, the graph package structure of the graph analytics is a consistent and modular solution to inserting advanced data processing into a pre-existing symbolic system. The structure leverages LOOPS' object-oriented properties and INTERLISP's symbolic properties to construct a clean and modular system. With direct object references, composable procedures, and layered organization, the system incorporates basic graph operations in a readable and editable manner. This early design not only verifies the research goals of this thesis but also serves as a basis for further refinement and instructional application in the INTERLISP environment.

3.3 Implementation of Core Components

The use of the graph package analysis in INTERLISP LOOPS is about a large body of base functions, all directly defined with no reliance on packaged ones. Organization is one emphasizing clarity, modularity, and symbolic form. Overall organization leans strongly on the object system of INTERLISP and facilities for list manipulation in constructing and analyzing graph structure. This chapter includes

a thorough explanation of the main components and algorithms, along with example pseudocode and implementation information.

3.3.1 Graph Construction and Modification

The first step toward enabling graph analytics was to define basic functions that add, remove, and update the graph structure. A few of these include *addVertex*, *removeVertex*, *addEdge*, and *removeEdge*. Vertices and edges are stored as internal lists of the Graph object, and updates are handled by modifying these lists in-place. Below is the pseudocode for these operations:

```
Procedure addVertex(graph, vertex):
    graph.vertices ← vertex :: graph.vertices

Procedure addEdge(graph, edge):
    graph.edges ← edge :: graph.edges

Procedure removeVertex(graph, vertex):
    graph.vertices ← REMOVE(vertex, graph.vertices)

Procedure removeEdge(graph, edge):
    graph.edges ← REMOVE(edge, graph.edges)
```

This implementation keeps the graph mutable with linear-time deletion and constant-time insertion, which is sufficient for the size and complexity of graphs expected in this environment. Duplicate vertices (same reference) are not present, and edges are treated to be unique based on source-target relationship.

3.3.2 Breadth-First Search (BFS) and Pathfinding

The primary graph traversal and pathfinding method implemented is Breadth-First Search, employed as a basic graph traversal as well as to determine shortest paths for unweighted undirected graphs.

The *bfs* function takes as input a start vertex and procedurally visits every reachable vertex with distances from the start node. It outputs a list of tuples for every reachable vertex and the distance to the origin.

ALGORITHM 1: Breadth-First Search

```
Procedure bfs(graph, start):
  visited ← [start]
  queue ← [(start, 0)]
  distances ← [(start, 0)]

  while queue is not empty:
    (node, dist) ← dequeue(queue)
    for neighbor in neighbors(node):
      if neighbor not in visited:
        enqueue(queue, (neighbor, dist + 1))
        visited ← visited + neighbor
        distances ← distances + (neighbor, dist + 1)

  return distances
```

This is the foundation for *allPairsShortestPaths*, which performs BFS from every vertex in order to compute shortest paths among all pairs of nodes, and *shortestPaths*, which computes the actual shortest path(s) between two vertices by attempting all possible paths of shortest length.

3.3.3 Centrality and Structural Measures

The package includes several functions to calculate network metrics and centrality scores:

- **Degree**: Counts immediate connections per vertex.
- **Closeness Centrality**: Compares inverse average shortest path length from one node to all others.
- **Betweenness Centrality**: Compares the frequency with which a node falls on shortest paths between others.
- **Eigenvector Centrality**: Estimated through recursive neighbor influence iterations.
- **PageRank**: Modeled through simplified iterative voting procedure.
- **Power Centrality**: Compares vertex influence in terms of reachability power.

These operations use both structural information and traversal output. Most are based on *allPairsShortestPaths* function, and some iterate over subgraphs or filtered sets of vertices.

These algorithms are equivalent to the ones found in modern libraries like *NetworkX* or *igraph*, but are implemented from scratch in terms of INTERLISP list operations and LOOPS method dispatch. This testifies to the programmability of older symbolic environments for carrying out high-level computational analysis.

ALGORITHM 2: Betweenness Centrality

```
Procedure betweenness(graph, vertex):
  totalPaths ← 0
  passingPaths ← 0
  for each pair (s, t) in vertices:
    if s ≠ t and s ≠ vertex and t ≠ vertex:
      paths ← shortestPaths(graph, s, t)
      totalPaths += |paths|
      for path in paths:
        if vertex in path excluding ends:
          passingPaths += 1
  return passingPaths / totalPaths
```

Each algorithm operates on the object-based graph structure, retrieving vertex and edge information through accessors and calculating properties from the resulting relationships.

3.3.4 Graph Properties and Structural Checks

Along with node-level processing, several functions check the global graph structure:

- **isConnected**: Verifies that all nodes are reachable from any other node.
- **diameter**: Gives the maximum of the shortest distances.
- **density**: Computes the number of edges over vertices against the theoretical limit.
- **eccentricity**: Computes the maximum distance between a vertex and any other node.

These operations provide simple information about the shape and topology of the graph. They also provide the basis for more complex algorithms, e.g., clustering or community detection.

ALGORITHM 3: Graph Connectivity

```
Procedure isConnected(graph):
  for each pair (source, target):
    if shortestPaths(graph, source, target) is empty:
      return false
  return true
```

These utilities provide essential tools for understanding the graph's structural form and serve as foundations for interpreting more advanced analytical outputs.

3.3.5 Community Detection and Clustering

Advanced methods such as Label Propagation (*communities*) and Bron–Kerbosch (*cliques*) exist to discover clusters and communities in the network. They use iterative label updates or recursive backtracking to identify densely connected groups.

- **Label Propagation Algorithm**: Assigns cluster labels to nodes based on majority voting among neighbors.
- **Bron–Kerbosch Algorithm**: Finds maximal cliques using recursive backtracking.
- **Cross Clique Centrality**: Measures node importance in overlapping communities.

ALGORITHM 4: Label Propagation

```
Procedure communities(graph):
  Assign each vertex a unique label
  for a fixed number of iterations:
    for each vertex v:
      neighborLabels ← labels of neighbors
      v.label ← most frequent label in neighborLabels
  return groupings of vertices by final label
```

ALGORITHM 5: Finding Cliques

```
Procedure bronKerbosch(R, P, X, result):
  If P and X are both empty:
    Add R to result
    Return

  For each vertex v in P:
    newR ← R ∪ {v}
    newP ← P ∩ neighbors(v)
    newX ← X ∩ neighbors(v)
    bronKerbosch(newR, newP, newX, result)
    P ← P - {v}
    X ← X ∪ {v}

Procedure cliques(graph):
  R ← ∅
  P ← set of all vertices in graph
  X ← ∅
  result ← empty list
  bronKerbosch(R, P, X, result)
  Return result
```

These algorithms perform well on small- to medium-sized graphs and were chosen because they are computationally straightforward and symbolically transparent. Their coding is inspired by descriptions in contemporary literature (e.g., Neo4j Graph Algorithms) but re-stated in INTERLISP syntax and reasoning.

3.3.6 Utility Functions and Graph-Wide Analytics

The system provides functions to increase its analytical aspect:

- **mst**: Finds a Minimum Spanning Tree using BFS logic
- **eccentricity**: Finds the farthest distance of a node from any other
- **isBipartite**: Determines if a graph can be partitioned cleanly
- **graphEfficiency**, **contributionCentrality**, **crossCliqueCentrality**: Calculate sophisticated network metrics for robustness and impact

ALGORITHM 6: Check if the Graph is Bipartite

```
Procedure isBipartite(graph):
  color ← empty map
  For each vertex v in graph.vertices:
    If v not in color:
      queue ← [v]
      color[v] ← 0
      While queue is not empty:
        current ← dequeue(queue)
        For each neighbor in neighbors(current):
          If neighbor not in color:
            color[neighbor] ← 1 - color[current]
            enqueue(queue, neighbor)
          Else if color[neighbor] = color[current]:
            Return False
  Return True
```

This function checks whether the graph can be divided into two disjoint sets such that no two vertices in the same set are adjacent. The algorithm "colors" (0 and 1) alternate adjacent vertices in a Breadth-First Search fashion. If there is a conflict (two adjacent vertices have the same color), then the graph is not bipartite.

ALGORITHM 7: Graph Efficiency

```
Procedure graphEfficiency(graph):
  total ← 0
  count ← 0

  For each pair of distinct vertices (u, v):
    If u ≠ v:
      paths ← shortestPaths(graph, u, v)
      If paths is not empty:
        shortest ← length of the first path in paths
        efficiency ← 1 / shortest
        total ← total + efficiency
        count ← count + 1

  If count > 0:
    Return total / count
  Else:
    Return 0
```

Graph efficiency is the inverse average shortest path length among all unique pairs of vertices. It gives us insight into how fast information spreads through the network.

ALGORITHM 8: Contribution Centrality

```
Procedure contributionCentrality(graph, targetVertex):
  Initialize total ← 0
  Initialize contribution ← 0

  For each pair of distinct vertices (u, v) in graph:
    If u ≠ v and u ≠ targetVertex and v ≠ targetVertex:
      paths ← shortestPaths(graph, u, v)
      numPaths ← count(paths)
      If numPaths = 0:
        Continue
      numThroughTarget ← 0
      For each path in paths:
        If targetVertex ∈ path (excluding u and v):
          numThroughTarget ← numThroughTarget + 1
      contribution ← contribution + (numThroughTarget / numPaths)
      total ← total + 1

  If total > 0:
    Return contribution / total
  Else:
    Return 0
```

This algorithm cycles through all unordered vertex pairs, excluding the target vertex as an endpoint. For every pair:

- It calculates all shortest paths.
- It counts the number of those paths with the target vertex as an in-between one.
- The ratio of such paths (with the vertex) is retained.
- Finally, it returns the average contribution of the vertex across all pairs.

These capabilities fill in the scope of the system, giving users all the different ways of analyzing, interpreting, and interacting with graph data in the legacy environment.

Table 3.1: Initially Planned Functionality

Function	Description	Implemented
add/remove vertices and edges	Structural mutation	Yes
degree(v)	Computes the degree (number of connections) of vertex **v**.	Yes
neighbors(v)	Returns a list of all neighboring vertices of vertex **v**.	Yes
is_connected()	Checks if the graph is connected (if there is a path between any two vertices).	Yes
shortest_path(source, target)	Computes the shortest path between the **source** and **target** vertices.	Yes
betweenness(v)	Computes the betweenness centrality of vertex **v**.	Yes
closeness(v)	Computes the closeness centrality of vertex **v.**	Yes
eigenvector_centrality()	Computes the eigenvector centrality for all vertices in the graph.	Yes
page_rank()	Computes the PageRank of each vertex in the graph.	Yes
clustering_coefficient(v)	Computes the local clustering coefficient for vertex **v**.	Yes
clusters ()	Identifies and returns all connected components in the graph.	Yes
minimum_spanning_tree()	Finds the minimum spanning tree of the graph.	Yes
graph.density()	Computes the density of the graph (ratio of edges to possible edges).	Yes
diameter()	Computes the diameter of the graph (longest shortest path between any two vertices).	Yes

Table 3.1 gives the first list of core capabilities that was planned to be implemented in the graph analytics package. They are comprised of structural primitive operations, traversal algorithms, and some classical graph measures such as centrality, clustering, and path-related calculations. Each of the listed functions was created and successfully integrated into the system. It is worth noting, however, that the package is not limited to all these functions alone — many additional utilities and analytical procedures have since been implemented as the project evolved, which expanded the capabilities of the package outside the original context.

This section demonstrates the richness and applicability of the graph analysis system created in INTERLISP LOOPS. Despite the symbolic and historical nature of the language, a wide range of analytical functions, ranging from straightforward traversal to complex structure analysis, have been realized. Object-oriented programming, symbolic computation, and dynamic list manipulation all worked together to produce a rich and versatile toolset similar to modern-day graph libraries. The implementation proves the continued validity of INTERLISP to designing modular and extensible computational software even in this day and age of modern languages.

3.4 Testing and Validation Strategy

Correctness, reliability, and consistency of the graph package were validated via an iterative process of testing and verification within the INTERLISP environment. Due to the systems nature — executed in the earlier symbolic programming environment without modern tools support for validation — particular care was taken within the design of manual test cases, observation-oriented verification procedures, and logging to ensure correctness. This section explains methods employed for testing basic functionality, result checking, handling edge cases, and algorithmic output checking.

3.4.1 Testing Strategy

Testing procedure was primarily executed by executing individual functions on manually created test graphs and comparing their results against theoretical predictions. Step-by-step inspection, logging systems, and debugging output were used to track and confirm intermediate computations. Even though there was no explicit unit test suite or test suite executed automatically, correctness was tested by

matching output of graph operations with known solutions from book examples, small graph datasets, or known results produced by current graph libraries.

To enable this, a global flag variable called *debug* was introduced. In debug mode (set to T, which means true), the system produces detailed trace outputs during function calls. The output includes order of traversal, intermediate path segments, and internal graph states, providing a clear glimpse of the step-by-step operations of the algorithm. This was particularly useful when coding and optimizing algorithms such as breadth-first search, shortest path finding, and community detection.

Moreover, the system also provides logging capabilities which store analysis output and structural data about the graph into an output file. This file does not just record final function outputs but also metadata such as number of vertices and edges, connectivity status of the graph, etc. The logs serve a dual purpose as a manual validation record and documentation records for future package users or package developers.

3.4.2 Validation Approach

Without access to outside testing infrastructure, correctness was demonstrated by comparative validation. As an example, the *shortestPath* function was tested by comparing against known paths within simple test graphs (e.g., triangle graphs, line graphs). Outputs of functions such as *isConnected*, *diameter*, and *graphEfficiency* were cross-compared against theoretical estimates and reference implementations in modern languages. Centrality and clustering values were compared against identified graph structures to ensure that outcomes aligned with conceptual expectations (e.g., highest degree nodes ranking highest in degree or betweenness centrality).

This comparative approach, while informal, proved sufficient to demonstrate that the algorithms act as expected on a set of representative examples. The symbolic character of INTERLISP rendered it relatively straightforward to view intermediary values and inspect them in real time.

3.4.3 Test Graphs and Execution Environment

Several graphs were tried, ranging from small, hand-built ones to larger ones with up to 50 nodes. Some of the graphs used were:

- Fully connected graphs, to check for performance and centrality computations.
- Disconnected graphs, to check correctness of *isConnected* and shortest path routines.
- Star and tree structures, to check hierarchical relationships and path consistency.
- Sparse graphs, to check for performance in low-density situations.

Graphs may be constructed interactively by means of the *addVertex* and *addEdge* procedures or read from outside files in INTERLISP format. This flexibility enabled a combination of exploratory testing and structured validation.

Graph outputs were retained in all situations in either human-readable or symbolic form for further inspection.

3.4.4 Edge Case Handling

The design was well-suited to handle a variety of edge cases:

- No edges or vertices: All functions return sensible defaults or raise informative exceptions.
- Disconnected graphs: Connectivity and pathfinding functions correctly report unreachable nodes or partitions.
- Duplicate edges: The system is programmed not to add them by checking existing relationships before insertion.
- Invalid references (e.g., operations on non-existent nodes): These are handled with warnings or null returns, and not through crashing or undefined behavior.

By handling all functions in these conditions, the package demonstrates robustness and reliability suitable to both analytical and exploratory use.

3.4.5 Current Limitations

Although all of the main functions were tested manually, no test framework is currently automated in the package. Given the symbolic and interactive nature of INTERLISP, building a self-testing validation system — i.e., a suite of reusable test graphs and corresponding expected output matchers — would be a huge improvement for ensuring the long-term reliability and reproducibility of testing.

This remains an area for future research, especially for education or community-based development, where automated correctness checking would be both a learning experience and quality assurance.

In summary, the graph analytics package validation was achieved by combining manual testing, debug output inspection, and logging. Although operating in a legacy environment with no external tools, the method used was sufficient to guarantee correctness of basic functionality and analytical results. Future growth may include an official internal test infrastructure to automate and expand validation coverage, but the current implementation has been demonstrated to function properly on a wide range of input graphs and structural circumstances.

3.5 Development Workflow

Development of the graph analytics package for INTERLISP LOOPS was iterative and modular, allowing for flexibility of design and response to issues as they arose. While the initial plan defined specific phases of functionality—from graph representation through analytical operations—the process evolved organically through cycles of implementation, testing, and refinement. The methodological development sequence, primary phases, pivotal decision points, and tools enabling the process are outlined in this section.

3.5.1 Development Approach

The project started by defining and using the base data structures to model graph elements. The Graph, Edge, and Vertex classes were the structural foundations of the system. Once the base classes had been defined in the LOOPS object system, the first milestone was the definition of early structural methods—with add and delete edge and vertex functions, and internal graph consistency maintenance.

Once the building blocks were established, the next phase was to develop core pathfinding and traversal routines, such as breadth-first search and shortest path calculation. These were used as a foundation for additional sophisticated analytics and were required for graph model structural integrity testing.

The final and most advanced phase was the development of graph analytical functions, including centrality metrics, clustering algorithms, and global structure measures. These methods were founded on the earlier modules, typically reusing and extending lower-level functions in a composable and modular manner.

3.5.2 Iterative Implementation and Refinement

Although the developmental process had a rational beginning with structural components leading to analytics first, it remained iterative, nonetheless. Functions were being refined continuously once they were originally deployed, and potential weaknesses were found and rectified during operation or when building upon them. Traversal outputs were, for example, perfected to more accurately be utilized with other modules, and centrality algorithms were refined for performance.

This iteration cycle also allowed for loose adaptation as new needs arose. Several functions beyond the original project scope were designed and added after they were discovered to be useful while testing or deploying other features. Some functions originally completed, such as directed edge support, were removed or pushed to the side to provide a limited scope and core feature stability.

Debugging and testing were conducted both continuously during development as well as at fixed milestones. In normal development, outputs were visually inspected, and behavior of the functions was tested using small, hand-coded test graphs. Towards this end, a special debug mode was provided, allowing step-by-step output of algorithmic execution when invoked through a global flag (*debug*). This

proved to be important in tracing complex behaviors and checking correctness of traversal and pathfinding algorithms.

In addition, at major milestone points in development—such as upon the completion of all the traversal operations or the first set of analytics operations—more thorough testing was carried out. That included correctness checking for different graph topologies and ensuring that there were no regressions introduced into earlier modules.

The project was implemented primarily on Windows using the Medley INTERLISP environment but was also tested on macOS for portability check. The Visual Studio Code was used to control the implementation, with which INTERLISP code files were written and edited since they are familiar to use and come with support for structured editing.

INTERLISP documentation, including the LOOPS manuals and historical language references, was a key source for learning about syntax and object system semantics. INTERLISP code fragments from the manuals and web archives also provided guidance on the idiomatic use of symbolic and object-oriented features.

The overall development process was very adaptive. The original roadmap was revised many times because of practical limitations and design factors. For example, while the system was originally designed to support directed and undirected edges, it was determined to concentrate on undirected graphs first to simplify initial implementation and improve consistency among algorithms.

Emerging requirements—like the inclusion of various centrality measures, community detection, and global graph measures—were introduced on an incremental basis as the system matured. This flexibility is required in a legacy environment like INTERLISP, where there is no traditional tooling, and many implementation decisions need to be re-evaluated on an ongoing basis.

In conclusion, building the graph analytics package was done through a phased, iterative, and adaptive method. From initial data structure and core method construction, development evolved to higher levels of analytics and comprehensive testing. The process enabled variability in priority, included test feedback, and was guided by both technical necessity and symbolic programming exploratory character of INTERLISP. The exercise demonstrated the feasibility of INTERLISP for structured software development and showed the effectiveness of incorporating legacy environments into modern programming methodologies.

3.6 Summary

This chapter presented the overall outline of the design, implementation, and development cycle of the graph package built using INTERLISP LOOPS. The methodology centered on a modular and object-oriented structure, implemented from scratch using symbolic programming principles and the object system offered by LOOPS. Through a step-by-step iterative approach, the system was incrementally constructed: starting from graph construction facilities, followed by search algorithms like breadth-first search, and finally moving towards in-depth analytical features like centrality functions, community detection, and global structural analysis.

All core modules were formally verified using controlled experiments, correctness being demonstrated through output comparisons, debugging aids, and functional trace logs. Object-oriented design achieved extensibility and symbolic data structures achieved dynamic graph manipulation within the legacy environment.

By embracing a hands-on and bottom-up development approach, the study directly contributes to further advances in the ultimate goal of this thesis — demonstrating that legacy environments like INTERLISP can be repurposed for enabling contemporary computational needs, e.g., graph analytics, without the need for third-party dependencies. The methodology, although based on a legacy environment, is centered on transparent software engineering approaches, reusability, and flexibility — qualities most important for system modernization.

Having established the base system and validated it, the next chapter will present the package results and evaluation, including qualitative comments about its behavior as well as analysis of its analytical output across test cases.

4 RESULTS

The purpose of this chapter is to present the evaluation of the graph analytics package developed for INTERLISP LOOPS. The central goal of the evaluation is to confirm the functional correctness of the system — to ensure that all algorithms developed give correct and consistent results for an extensive variety of graph structures. Because the package was constructed from scratch in a legacy symbolic programming environment, internal consistency and correctness were more critical than performance benchmarking.

The evaluation is done in terms of ensuring that the system works as required in performing critical graph operations such as traversal, shortest path calculation, centrality computation, community detection, and structural analysis. To this end, a test suite was constructed, including both small, hand-declared graphs and larger graphs with up to 50 nodes. These graphs were chosen to demonstrate a variety of topologies — connected and disconnected graphs, sparse and dense networks, and representative substructures such as stars, cliques, and trees.

Besides functional accuracy, the testing also investigates other relevant properties of the system. These include the usability of the package, particularly within the constraints of INTERLISP's interactive programming model, and how readable and modular the analytics outputs are. Particular attention is also given to features such as the debugging and logging facilities, which were intended to facilitate usage as well as validation. They facilitate the user to interpret analytic results and provide insight into algorithmic processes — an important requirement in symbolic environments where step-by-step introspection takes a central role.

The remainder of this chapter demonstrates the results of applying core functions to sample test graphs and remarks on their accuracy, behavior, robustness, and usefulness. Thus, the capabilities of the package are demonstrated and its analytical applicability in INTERLISP is confirmed.

4.1 Testing Setup and Environment

The graph analytics software package was implemented and tested in a stable, controlled environment to deliver consistent and reproducible results. The primary platform used throughout the project was a Windows 11 machine on which the Medley INTERLISP environment was loaded and configured. Even though the Medley system has its roots in legacy computing, it was highly functional in this modern setup without the requirement for additional emulation or compatibility layers beyond the normal installation procedure.

The INTERLISP environment was run in its interactive mode to allow for real-time graph construction, function invocation, and result inspection. The entire testing occurred within the Medley interface, with LOOPS being used to define class structures and execute analytics algorithms. Editing of code was provided externally through Visual Studio Code (VSCode), where Lisp files were programmed and edited before loading the files into the INTERLISP system.

Graphs used for testing were constructed in two ways:

- **Manual graph creation**, where vertices and edges were added dynamically through *addVertex* and *addEdge* methods to test key operations incrementally and in a controlled way.
- **File-based graph input**, where graphs were defined in special text files as an adjacency list of vertices and loaded into INTERLISP at run time. Fixed-format files were used, generally listing vertices and the vertices which they were connected to. It was particularly useful for testing mid-size graphs or repeating test scenarios.

Figure 4.1 shows an example graph definition loaded from a plain text file to demonstrate the structure used to process file-based graph input.

Figure 4.1: Sample graph input file

In this format, each line is split at the colon (:) marker. The left-hand side of the colon is the vertex of origin, and the vertices it goes to, separated by spaces, on the right-hand side. For example, the line *A: B C D* indicates that vertex *A* goes to vertices *B*, *C*, and *D*.

When this file is being loaded into INTERLISP, the package does a dynamic check to determine if every vertex is already present in the graph. If a vertex hasn't been created yet, it's created on the fly and inserted into the internal list of vertices. In this manner, all the nodes that will be used are set up correctly prior to the creation of any edges. Connections are then established between each source vertex and its neighbors by creating undirected edges, unless there is already an existing edge connecting the same pair.

This format simplifies testing and batch input of larger graph structures, making reusable datasets and repeatable experiment setup possible during validation.

To facilitate validation, the system has a logging capability that saves key information about graph structures and algorithmic outputs to a plain text file. Logs include outputs, graph statistics, and step-by-step traces (when in debug mode). Logging enabled examination of outputs after execution and verification of correctness without having to re-run operations interactively. In addition, a global debug flag was added to control verbosity — when set to true, this mode enabled verbose printouts during runtime, making logic flow, internal states, and intermediate outputs more easily traceable.

This setup provided an interactive, open development/test environment appropriate to the symbolic and interactive nature of INTERLISP. The ability to switch to and from manual and automatic input, observe run-time behavior in real time, and inspect output files provided complete verification of each feature of a package in diverse kinds of graphs.

4.2 Functional Demonstration and Output Validation

4.2.1 Graph Construction and Traversal

To verify the internal correctness and validity of the graph building process, a test graph was created based on the input file structure described in Figure 4.1. It is a seven-vertex (A through G) graph with some number of undirected edges depicting connectivity between them. As described previously, each line in the input file is associated with mapping a vertex to the vertices to which it is connected. As part of parsing the input, new nodes are automatically instantiated if they haven't already been defined in the structure of the graph, and edges are also created accordingly in order to ensure complete bi-directional connectivity.

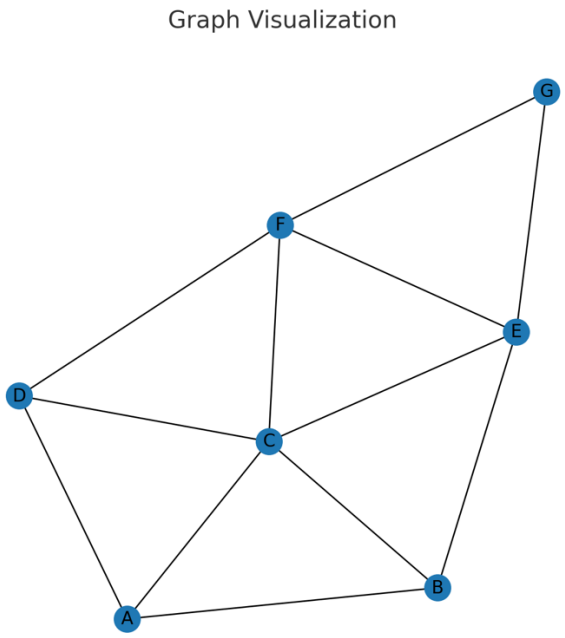


Figure 4.2: Visual representation of the test graph

A visual representation of this graph is shown in Figure 4.2, which was manually rendered for clarity. It shows the graph structure that was used as a test case for graph traversal operations.

Once the graph was constructed, the Breadth-First Search (BFS) algorithm was executed from various source vertices to validate the traversal algorithm. The BFS algorithm traverses the graph layer-by-layer and yields a list of discovered vertices in their discovery order together with their distances from the source vertex. The output is either printed straight on execution or stored in a log file depending on the environment mode.

One such result of the BFS function run with the starting node as vertex *A* is shown below in Figure 4.3.

```
("Visiting:" #,($ A) "Dist:" 0)
("  Found neighbor:" #,($ B) "-> Distance:" 1)
("  Found neighbor:" #,($ C) "-> Distance:" 1)
("  Found neighbor:" #,($ D) "-> Distance:" 1)
("Visiting:" #,($ B) "Dist:" 1)
("  Found neighbor:" #,($ E) "-> Distance:" 2)
("Visiting:" #,($ C) "Dist:" 1)
("  Found neighbor:" #,($ F) "-> Distance:" 2)
("Visiting:" #,($ D) "Dist:" 1)
("Visiting:" #,($ E) "Dist:" 2)
("  Found neighbor:" #,($ G) "-> Distance:" 3)
("Visiting:" #,($ F) "Dist:" 2)
("Visiting:" #,($ G) "Dist:" 3)
((#,($ G) 3) (#,($ F) 2) (#,($ E) 2) (#,($ D) 1) (#,($ C) 1) (#,($ B) 1) (#,($ A) 0))
```

Figure 4.3: Output of the BFS traversal from vertex *A*. Each line includes a vertex and its distance from the start

The output illustrates that the traversal correctly visits neighbors and only advances to the next level once all vertices at the current level are visited. That is, *B*, *C*, and *D* are distance 1 away from *A*; *E* and *F* are distance 2, and *G* is visited at distance 3. The outputs are consistent with the graph structure as graphically visualized.

The traversal algorithm works as expected in all tested conditions, including traversals starting from detached or leaf nodes. The outcome confirms that node finding, queue management, and handling cycles are all correctly achieved with INTERLISP's symbolic data structures. The BFS algorithm is also

a root algorithm for other algorithms, such as shortest path finding and centrality measures, depending on its outputs for computation.

4.2.2 Shortest Path and Distance Computations

Shortest path algorithms are arguably the most fundamental constructs of graph analytics, used for everything from routing networks to influence estimation. To provide this capability in the INTERLISP environment, single-pair and all-pairs shortest path algorithms were implemented and run on the constructed graph.

- **Single-Pair Shortest Path**

The *shortestPath* function computes the shortest path from any source vertex to target vertex by doing a breadth-first search. It returns the shortest path as a list of vertices, in order visited. A sample call to this function is illustrated in Figure 4.4, which shows output from running (*shortestPath G1 A G*), with test graph *G1*, source *A* and target *G*.

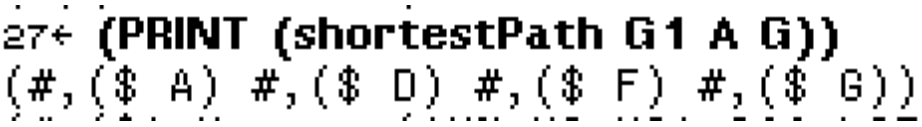


Figure 4.4: Output of the *shortestPath* function for vertices A and G

The result is a list of vertices for the path from *A* to *G*: $A \rightarrow D \rightarrow F \rightarrow G$. This is the shortest unweighted path in the graph and demonstrates that the BFS-based solution correctly identifies the minimum route by the number of edges.

- **All-Pairs Shortest Paths**

To include this functionality, *allPairsShortestPaths* was written to calculate shortest path distances between all vertices to all others. The return value is a list where each tuple is the source vertex and a sublist of destinations and their associated distances. A run of the function on the same graph is shown in Figure 4.5.

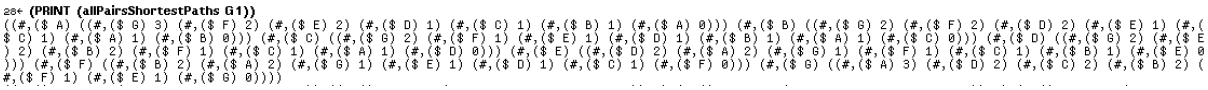


Figure 4.5: Output of *allPairsShortestPaths* displaying distances between all vertex pairs

The graph will thus clearly show each vertex's shortest distances to all other reachable nodes. For example, vertex *A* is 3 from *G*, as would be predicted from the single path calculated above. Non-linked nodes are not included in the output, preserving efficiency and eliminating null results.

The outputs of both operations were also validated against theoretical expectations with the given test graph structure (as described in Figure 4.2). In both cases, the outputted distances and paths were correct and complete, establishing the validity of the traversal and pathfinding logic.

Moreover, the consistency of single-pair and all-pairs results guarantees internal reliability of the package. They are the basis of many other analytics components — centrality, diameter, and efficiency — and their accuracy is the foundation of the whole system's integrity.

4.2.3 Centrality and Structural Metrics

Centrality measures are crucial in identifying influential vertices within a graph and the underlying structure. The package contains implementations for most of the common centrality measures, such as degree, betweenness, closeness, eigenvector, and PageRank centralities, and structural measures such as graph density, diameter, and connectivity. These procedures were tested on the same test graph employed in earlier sections and produced correct, interpretable outputs.

Every function was designed to take a graph and target vertex (where applicable) and return a numeric score as a measure of the vertex's position in the network. Results were printed to the console immediately or logged and thus accessible in interactive development and offline validation phases.

A vertex's degree centrality function determines how many direct connections it has. The example graph's central nodes regularly produced the highest degree values and this was expected. Because the values could be visually verified against the graph structure, this acted as a helpful sanity check.

```
31> (degree A)
3
32> (degree B)
3
33> (degree C)
5
34> (degree D)
3
35> (degree E)
4
36> (degree F)
4
37> (degree G)
2
```

Figure 4.6: Output of the *degree* function showing the number of direct connections for each vertex

The frequency with which a vertex appears on the shortest paths between other vertices was measured by the betweenness centrality function. Vertices along bridges connecting clusters in the test graph scored higher. The result displays their importance in preserving connectivity.

```
38> (betweenness G1 A)
0.05263158
39> (betweenness G1 B)
0.09090909
40> (betweenness G1 C)
0.3043478
41> (betweenness G1 D)
0.09090909
42> (betweenness G1 E)
0.22727273
43> (betweenness G1 F)
0.22727273
44> (betweenness G1 G)
0.0
```

Figure 4.7: Output of the *betweenness* function indicating the frequency of each vertex appearing on shortest paths

The values in Figure 4.7 agree with the observation that vertex *C* has the highest betweenness value (approximately 0.30), which means that it plays a central role in connecting various parts of the graph. This agrees with the graph structure where *C* is a central vertex connecting various regions. Vertices *E* and *F* have fairly high values, which is a sign that they play a role in communicating between other nodes. Vertex *G*, though, is given a score of 0.0, since it is a peripheral node that only has connections to *E* and *F*, and never appears on any shortest path between other pairs. These results validate both the correctness of the algorithm and the interpretability of the package's output.

The closeness centrality algorithm calculated the reciprocal of the average shortest path from a node to all other nodes. Nodes near the center of the graph received higher scores, and peripheral nodes such as G returned lower values, as one would expect.

```
45> (closeness G1 A)
0.6
46> (closeness G1 B)
0.6666667
47> (closeness G1 C)
0.85714287
48> (closeness G1 D)
0.6666667
49> (closeness G1 E)
0.75
50> (closeness G1 F)
0.75
51> (closeness G1 G)
0.54545456
```

Figure 4.8: Output of the *closeness* function, reporting proximity of each node to all others in the graph

In addition, eigenvector centrality and PageRank centrality were developed using iterative algorithms. Both centralities point to nodes connected to influential nodes, and both give a subtle influence in the network. These measures showed noticeable scores for different nodes regardless of the graph size being small.

```
52> (eigenvectorCentrality G1 A)
0.12815918
53> (eigenvectorCentrality G1 B)
0.13375546
54> (eigenvectorCentrality G1 C)
0.19716528
55> (eigenvectorCentrality G1 D)
0.13375546
56> (eigenvectorCentrality G1 E)
0.15956594
57> (eigenvectorCentrality G1 F)
0.15956594
58> (eigenvectorCentrality G1 G)
0.08803272
```

Figure 4.9: Output of the *eigenvectorCentrality* function, showing influence scores based on connection to important nodes

```
59← (pageRank G1 A)
0.12679595
60← (pageRank G1 B)
0.1262799
61← (pageRank G1 C)
0.19905147
62← (pageRank G1 D)
0.1262799
63← (pageRank G1 E)
0.16500781
64← (pageRank G1 F)
0.16500781
65← (pageRank G1 G)
0.09157714
```

Figure 4.10: Output of the *pageRank* function representing node importance using an iterative ranking approach

In general, graph diameter and density served well to describe the structure of the graph. For example, the diameter function that produced the size of the longest shortest path correlated with BFS-calculated values, and the density indicated how close to a completely connected graph the graph was.

```
66← (diameter G1)
3
67← (density G1)
0.5714286
```

Figure 4.11: Outputs of the *diameter* and *density* functions

In general, the output of these operations supports the power and breadth of analysis capabilities that the system possesses. The output tracked very well with the known topology of the test graphs, and no unexpected anomalies or errors were noted. These structural and centrality functions form building blocks of analyses further developed and support the package as applicable to graph exploration in real-world situations.

4.2.4 Community Detection and Clique Identification

Subgroup identification and community structure are key elements in understanding the topology and social structures of complex graphs. Several algorithms included in the package address these problems, which were tested and validated separately by applying them to graphs with clear as well as subtle groupings.

- **Connected Components**

The *clusters* function determines connected components in the graph through a breadth-first search. Each component is returned as a list of vertices mutually reachable from one another, dividing the graph into structurally isolated regions. This facility proved useful in checking the disconnection behavior of the graph, ensuring that the algorithm correctly marks isolated subgraphs.

```
69← (PRINT (clusters G1))
((#,($ A) #,($ D) #,($ C) #,($ B) #,($ F) #,($ E) #,($ G)))
```

Figure 4.12: Output of the *clusters* function showing detected connected components in the graph

In the test graph, the function properly listed one connected component, since all of the vertices are reachable by some path. In other test graphs with disconnected organization, there were enumerated more than one component, thereby confirming the correctness of the clustering logic.

To show that this implementation is effective, another version of the test graph was built where two new nodes (*H* and *I*) were added as an independent component with only mutual edges to each other and no edge to the rest of the network (Figure 4.13).

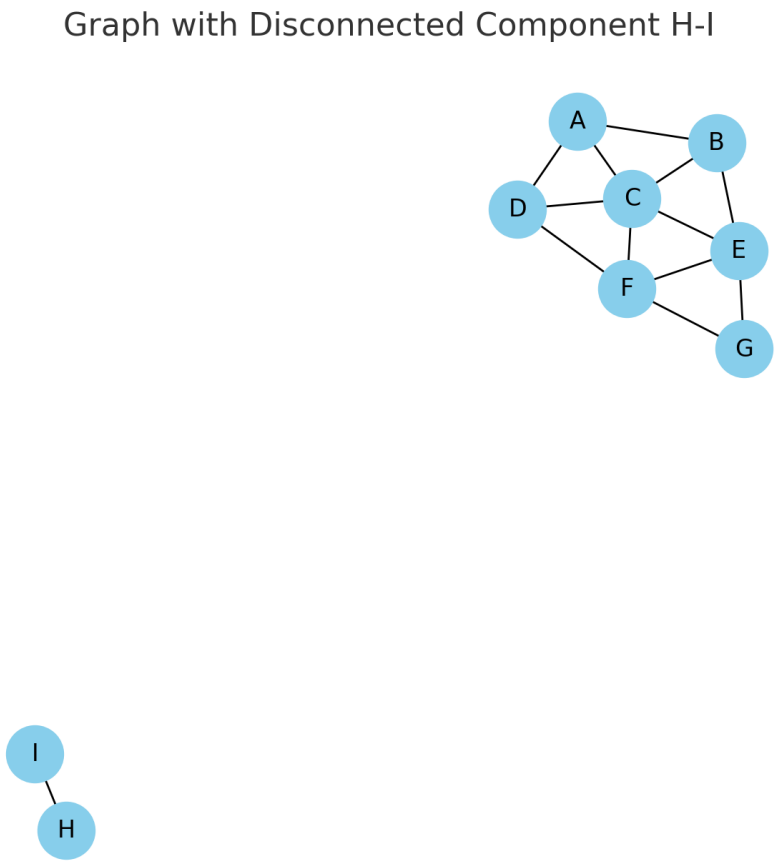


Figure 4.13: Modified graph with an added disconnected component

Operating the *clusters* as functions on this new graph created two distinct components — the major cluster of vertices A through G, and another component consisting of *H* and *I*. The output function is represented in Figure 4.14 and correctly models the underlying structure.

```
72← (PRINT (clusters G1))
((#,($ A) #,($ D) #,($ C) #,($ B) #,($ F) #,($ E) #,($ G)) (#,($ H) #,($ I)))
```

Figure 4.14: Output of the *clusters* function showing two distinct connected components in the graph

This result confirms the function to correctly detect isolated regions regardless of graph size and component complexity. It is in accordance with the traversal algorithm used in the function and supports the robustness of the implementation in the event of more fragmented topologies.

- **Community Detection**

The *communities* function uses a label propagation method. Each node is first assigned a distinctive label. At a predetermined number of iterations, each node takes over the most prevalent label of its neighbors. In the process, this results in convergence to influential local clusters.

```
75← (PRINT (communities G1))
((#,($ G) #,($ F) #,($ E) #,($ D) #,($ C) #,($ B) #,($ A)))
```

Figure 4.15: Output of the *communities* function indicating detected groups based on neighborhood label agreement

This algorithm well clumped heavily interconnected nodes together. Empirically, simple graph results were deterministic, though results in situations with more than one similarly frequent label might see some variance in across runs.

- **Clique Detection**

To discover maximal cliques — subsets of vertices with each node directly connected to every other node — the package uses the recursive *Bron–Kerbosch* algorithm as the *cliques* function. The function delivers all of them in the graph, each being a complete subgraph.

```
77← (printCliques (cliques G1))
("Clique:" #,($ B) #,($ A) #,($ C))
("Clique:" #,($ C) #,($ A) #,($ D))
("Clique:" #,($ C) #,($ B) #,($ E))
("Clique:" #,($ D) #,($ C) #,($ F))
("Clique:" #,($ E) #,($ C) #,($ F))
("Clique:" #,($ F) #,($ E) #,($ G))
```

Figure 4.16: Output of the *cliques* function showing all maximal fully connected subgraphs

The test graph had several 3-node cliques, and all were correctly reported. This proves the implementation's ability to navigate the structure of the graph recursively and return complete sets without redundancy.

Together, these functions create a powerful toolkit for structural analysis of graphs. Their findings were consistent with theoretical predictions and provided more information than centrality or traversal. They are required to find internal groupings and symmetry of more complex networks.

4.3 Robustness and Error Handling

Ensuring that a system behaves in a predictable way under both normal and faulty conditions is most important to its reliability. In addition to testing the correctness of basic functions, the graph analysis package was also tested for robustness by running different edge cases and invalid input scenarios.

The system is gracious in dealing with empty graphs. When there are no vertices or edges, operations such as traversal, centrality, or connectivity tests return appropriate default values — e.g., empty lists or null values — without resulting in runtime errors. This is an assurance that base cases were considered during development and that base functions can operate on minimally defined graphs.

Duplicate insertion attempts are also dealt with effectively. When a user tries to insert a vertex that has already been inserted, the system silently disregards the attempt without disrupting the graph structure. Similarly, if an edge between two vertices has already been inserted, further efforts to insert the same edge fail. This prevents undesirable duplication and maintains the integrity of the internal data model and no explicit validation operations are required from the user.

Where graph operations are performed on disconnected structures, e.g., shortest path or community detection operations, the system produces incomplete results or skips inaccessible nodes as suitable. Pathfinding between two vertices in different connected components would return no path, e.g., respecting the constraints imposed by the graph topology. The system is therefore able to keep function outputs meaningful and consistent even under non-full connectivity.

The system does not yet include direct input validation for non-existent vertices or edges. When a function is called with an incorrect vertex label, INTERLISP gives a system error message that the object is not present. While these errors are not caught and handled in the system, they are consistent

with INTERLISP environment behavior and usually are informative enough to enable programmers to correct their input.

Although no specific exception-handling constructs or error wrappers were employed, the stability of the package in common edge cases is a high level of robustness. With symbolic debuggers and an optional verbose log mode, this equates to users being provided with insight into problematic input cases and being able to trace faults during development as well as deployment.

4.4 Usability and Debugging Features

Ease of use of analysis tools is often just as important as computational power, especially in a setting like INTERLISP, where development and debugging workflows differ so much from those in modern programming environments. With this in mind, the graph analysis package was developed with ease of use, readability of output, and flexibility of function invocation as primary concerns. In this chapter, the various features that allow for user interaction with the package, graph manipulation, and debugging and evaluation efficiently are discussed.

The package supports two principal modes of graph construction: file-based and manual. Under manual construction, users operate on functions such as *addVertex* and *addEdge*, via which they can construct incrementally in real time the structure of a graph. This construction mode is particularly helpful in experimentation or pedagogy, where the user may desire to create and modify incrementally stepwise a graph in order to observe how modifications affect its properties.

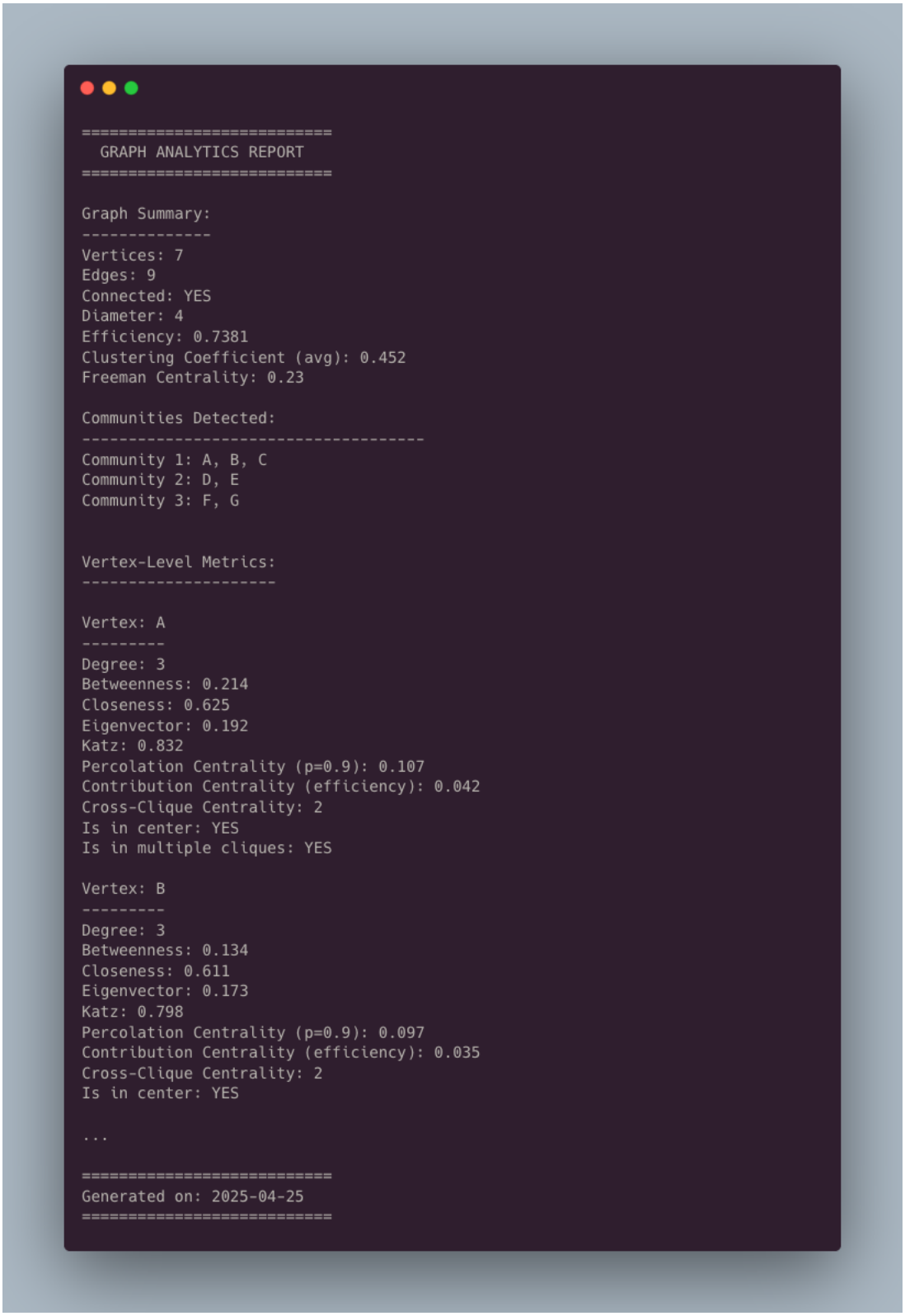
Concurrently, a file-based input mode was introduced to support larger or pre-defined sets of data. Users can also encode graph structure as plain text where a vertex is encoded in its neighbors separated by a colon (:) separator. The system reads this and it constructs vertices and edges as needed. Such flexibility allows seamless switching between interactive development and reproducible testing.

Once the graph has been constructed, users may invoke single analytics procedures (e.g., betweenness, closeness, or clusters) or execute a comprehensive wrapper function that conducts all major analyses and writes them out into a report file.

The package focuses on human-readable output for convenient interpretation and sharing of results. Outputs are printed to the console in interactive use and to structured text files for reference later. A standard output report includes:

- Global properties such as vertex and edge number, connectivity status, graph diameter, and average clustering coefficient
- Community detection results showing the clustering of nodes
- Vertex-level measurements such as degree, some centralities, and clique membership status
- A timestamp of when the report was generated
- And many other metrics.

An example report is shown in Figure 4.17, which demonstrates the tidy format and rich information included in a typical analytics summary.



```
==========================
  GRAPH ANALYTICS REPORT
==========================

Graph Summary:
--------------
Vertices: 7
Edges: 9
Connected: YES
Diameter: 4
Efficiency: 0.7381
Clustering Coefficient (avg): 0.452
Freeman Centrality: 0.23

Communities Detected:
-------------------------------------
Community 1: A, B, C
Community 2: D, E
Community 3: F, G


Vertex-Level Metrics:
---------------------

Vertex: A
---------
Degree: 3
Betweenness: 0.214
Closeness: 0.625
Eigenvector: 0.192
Katz: 0.832
Percolation Centrality (p=0.9): 0.107
Contribution Centrality (efficiency): 0.042
Cross-Clique Centrality: 2
Is in center: YES
Is in multiple cliques: YES

Vertex: B
---------
Degree: 3
Betweenness: 0.134
Closeness: 0.611
Eigenvector: 0.173
Katz: 0.798
Percolation Centrality (p=0.9): 0.097
Contribution Centrality (efficiency): 0.035
Cross-Clique Centrality: 2
Is in center: YES

...

==========================
Generated on: 2025-04-25
==========================
```

Figure 4.17: Sample text output generated by the analytics wrapper function

This output is not only a record of a specific analysis run but also a debugging tool — providing instant insight into whether specific values or structures are computed as expected.

To further verify the robustness and scalability of the analytics package developed, a more complex graph of 50 vertices was constructed and analyzed.

Figure 4.18 shows this graph, which is indicative of the capacity of the system to handle larger, more densely connected networks. The global wrapper function was executed on this dataset, generating a complete analytics report as with the smaller graph but reflective of the increased structural complexity.

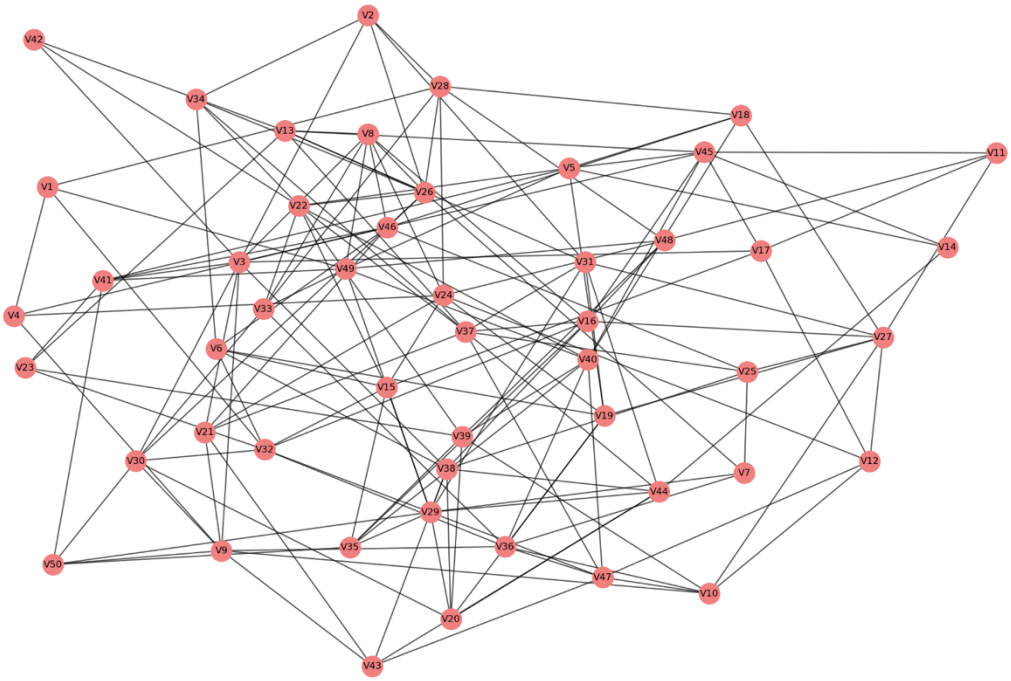


Figure 4.18: Complex graph with 50 vertices

In this graph, vertices and edges were generated randomly, with a non-trivial connection pattern that would put traversal algorithms, centrality calculations, clustering, and structural metrics through their paces.

As with the smaller graphs, analysis results were written to a plain-text file format, keeping both graph-level and vertex-level metrics for analysis.

The structural examination of the complex network revealed interesting structural properties. Core hub nodes responsible for high connectivity of the network were well differentiated using betweenness and eigenvector centralities. The clustering coefficient analysis indicated that there existed moderate local cohesiveness of sets of nodes, which contained non-random community structures even for a randomized network. Moreover, traversal-based operations such as shortest paths and all-pairs shortest paths performed well, with diameter and average path length consistent with expected performance for graphs of these size and density. These outcomes confirm the developed analytics package to be performing well beyond simple graphs, functionality and interpretability still present even when dealing with more complex and interlinked structures.

In order to support incremental testing and internal validation, the system supports an overall debug switch controlled by variable *debug*. By setting *debug* as *T* (true), additional feedback is displayed upon execution of functions. They include:

- Incremental step-by-step progress in the traversal algorithms
- Messages in constructing graphs during verification
- Identification of edge cases (e.g., operations for appending duplicate vertices and edges)
- Intermediary computation such as queues temporarily, marked nodes, and recursion states

This verbosity mode is invaluable during development and tuning periods, especially during the debugging of logic bugs or verifying iterative algorithms behave as desired.

The debug mode may be enabled at any point while executing, so the user can switch back and forth between production-quality output and greater explanatory function tracing without affecting any of the underlying logic. This is about separation guarantees debugging facilities exist without corrupting output during normal operation.

```
78← (SETQ debug T)
T
79← (eigenvectorCentrality G1 A)
("Iteration" 0 "scores:" ((#,($ G) 0.083333336) (#,($ F) 0.16666667) (#,($ E) 0.16666667) (#,($ D) 0.125) (#,($ C) 0.20833333) (#,($ B) 0.125) (#,($ A) 0.125))
)
("Iteration" 1 "scores:" ((#,($ G) 0.09090909) (#,($ F) 0.1590909) (#,($ E) 0.1590909) (#,($ D) 0.13636364) (#,($ C) 0.19318183) (#,($ B) 0.13636364) (#,($ A)
0.12499999)))
("Iteration" 2 "scores:" ((#,($ G) 0.08805031) (#,($ F) 0.16037737) (#,($ E) 0.16037737) (#,($ D) 0.13207546) (#,($ C) 0.19811317) (#,($ B) 0.13207546) (#,($ A
) 0.12893082)))
("Iteration" 3 "scores:" ((#,($ G) 0.08838823) (#,($ F) 0.1594454) (#,($ E) 0.1594454) (#,($ D) 0.13431543) (#,($ C) 0.19670713) (#,($ B) 0.13431543) (#,($ A)
0.12738301)))
("Iteration" 4 "scores:" ((#,($ G) 0.08799617) (#,($ F) 0.1597322) (#,($ E) 0.1597322) (#,($ D) 0.13342898) (#,($ C) 0.19727403) (#,($ B) 0.13342898) (#,($ A)
0.12840746)))
("Iteration" 5 "scores:" ((#,($ G) 0.0881034) (#,($ F) 0.15952253) (#,($ E) 0.15952253) (#,($ D) 0.13386968) (#,($ C) 0.19711156) (#,($ B) 0.13386968) (#,($ A)
 0.12800051)))
("Iteration" 6 "scores:" ((#,($ G) 0.08800843) (#,($ F) 0.15960853) (#,($ E) 0.15960853) (#,($ D) 0.13368623) (#,($ C) 0.1971731) (#,($ B) 0.13368623) (#,($ A)
 0.12822892)))
("Iteration" 7 "scores:" ((#,($ G) 0.0880464) (#,($ F) 0.15955526) (#,($ E) 0.15955526) (#,($ D) 0.13377555) (#,($ C) 0.19716112) (#,($ B) 0.13377555) (#,($ A)
 0.12813088)))
("Iteration" 8 "scores:" ((#,($ G) 0.08802111) (#,($ F) 0.15957977) (#,($ E) 0.15957977) (#,($ D) 0.13373671) (#,($ C) 0.19716313) (#,($ B) 0.13373671) (#,($ A
) 0.12818281)))
("Iteration" 9 "scores:" ((#,($ G) 0.08803272) (#,($ F) 0.15956594) (#,($ E) 0.15956594) (#,($ D) 0.13375546) (#,($ C) 0.19716528) (#,($ B) 0.13375546) (#,($ A
) 0.12815918)))
0.12815918
```

Figure 4.19: Sample debug output for *eigenvectorCentrality*, showing the evolution of vertex scores across 10 iterations.

The output in Figure 4.19 shows the behavior of the eigenvector centrality function when run in debug mode. At each iteration, the algorithm updates each vertex's centrality score based on neighboring vertices' scores. The printout allows the developer to observe the convergence of scores to stable values, verifying both the implementation's correctness and its convergence properties. By showing iteration-level detail, the debug mode serves as an internal audit tool that makes it easy to verify the correctness of devious recursive or iterative logic.

Although the package does not have explicit error handling wrappers, it is resilient against common user errors. For instance, when a user attempts to add an existing vertex or edge, the system quietly disregards the action. This avoids structural inconsistency without hindering the user from sending duplicate requests at no cost.

When faulty inputs are entered — such as requesting analytics from a non-existent vertex — the INTERLISP environment answers with explanatory errors (such as unbound variable messages), which direct the user towards repair. These environment-level error messages are intuitive to users working in INTERLISP and generally are sufficient in most instances.

The system also has utility routines for graph inspection and maintenance, such as:

- Listing all vertices in the graph
- Printing neighbors of a vertex
- Checking if two vertices are adjacent
- Retrieving a textual summary of the entire graph structure

These utilities improve usability, especially for novices to INTERLISP or for users who may be experimenting with new data sets.

Another usability consideration is the ease with which the package can be scripted and automated. Due to the fact that each function has a uniform naming convention and argument convention, users can easily create scripts that generate graphs, run full analytics pipelines, and generate results in batch mode. The codebase modularity makes this extensibility possible, where users can add or remove functions from the analytics pipeline with minimal modification to the wrapper function.

This technique has proven particularly useful when doing testing, whereby different graph structures are read from file, piped through the analytics toolkit, and analyzed utilizing the reports produced — all without requiring repeated human intervention.

Finally, the system is built employing clear name conventions and inline comments so that users who wish to alter or augment it can do so. Each function has a header comment describing its usage, expected inputs, and output. As needed, implementation commentary is added in the interest of

describing edge cases or algorithmic assumptions. All this documentation is an extension of the overall goal to create a package not only useful, but also pedagogical and sustainable.

4.5 Summary of Results

The testing of the graph analytics package revealed that the system meets its most important design objectives: offering comprehensive analytical power, being operable within the INTERLISP environment, and producing interpretable and verifiable results. Through the testing of basic algorithms, structural analysis methods, centrality, and substructure detection, the package was demonstrated to be robust and functionally sufficient.

All the operations initially intended — such as traversal, shortest path, degree-based measures, centralities, connectivity tests, and community detection — were successfully executed and validated. The implementations' correctness was confirmed using controlled test graphs of varying complexity. In all cases, the results agreed with theoretical predictions and known benchmarks. Operations such as betweenness and eigenvector centrality showed considerable variation across nodes, validating the internal calculations. Contrary to this, clique detection and cluster detection yielded accurate results even for graphs that contain disconnected parts or dense clusters.

From a usability perspective, the package was designed to be compatible with both interactively created graphs and file-based inputs. The two-mode flexibility enabled simple testing, extension, and reproducibility of results. The availability of a human-readable logging mechanism, as well as a general text-based report generation feature, enabled users to obtain holistic summaries of graph-level and vertex-level properties. The inclusion of a debug flag also helped with internal verification using step-by-step output and iteration computation, for example, in the eigenvector centrality result.

One of the best achievements of the system is to provide these facilities completely within the INTERLISP environment without external verification or reliance on contemporary tools. This intrinsic approach demonstrates the potential of using a modern-day graph analytics scenario in a legacy system with the help of symbolic and object-oriented features native to INTERLISP LOOPS.

Although the development process revealed no major functional inadequacies, it did reveal some of the inherent challenges in supporting a legacy language — i.e., poorly documented information and the need to implement even basic data structures in a custom fashion. These problems were, however, solved to success, and by doing so the project lays groundwork for future extendability.

Overall, the results confirm that the graph package is not only correct in performance but also handy, modular, and extensible. It is a tool set for general graph exploration available to INTERLISP programmers and an effective demonstration of how modern algorithmic ideas may be enhanced within the symbolic computer environment. The project demonstrates that with thoughtful adaptation and design, legacy systems are still able to fulfill contemporary computational requirements to offer functional use and historical continuity in software studies.

5 SUMMARY AND FUTURE WORK

This project sought to develop and integrate a complete graph analysis package within the INTERLISP LOOPS environment — something that, to the best of available knowledge, had not yet been done. This project completed a longstanding gap in this historically significant language's environment by offering a stand-alone object-oriented tool for performing contemporary graph analysis operations via symbolic computation. Through careful design, incremental growth, and rigorous testing, the package has evolved into a robust and reusable system that bridges the capabilities of traditional Lisp environments with contemporary algorithmic demands.

Among the greatest achievements of this project is its originality. There has not been a previous realization of a graph analytics package in INTERLISP, let alone one that takes full advantage of LOOPS and includes such a great variety of centrality metrics, structural inquiries, and subgraph analysis functions. This renders the system not just operational but also historically significant as a baseline for future development within legacy systems.

Other key contributions include:

- A modular, reusable framework, where underlying structural and algorithmic primitives can be specialized or extended.
- Open integration with the symbolic processing and object-oriented capabilities of INTERLISP using LOOPS, reflecting the flexibility of the environment.
- Implementation of well-known analytics algorithms such as betweenness, closeness, eigenvector centrality, community detection, and clique finding, tested using extensive test cases.
- Usability-driven design, with organized console outputs, detailed report generation, debug toggling, and graph inspection/validation tooling.

Together, these contributions demonstrate that even legacy systems — often considered too constrained or limited for contemporary computational work — can be revitalized through diligent engineering and modern design principles.

While the package is functional and complete, there are directions waiting to be extended. Perhaps most importantly, the current system supports undirected graphs only, and the majority of actual networks are actually directional (e.g., citation networks, flow systems, web links). To include the data model to support directed edges, and to extend algorithms like PageRank or strongly connected components to this topology, would be an obvious extension.

Another area is the integration of visualization functionality. Currently, graph topologies and measures are accessed textually via console output or reported generation. Visualization of graph topologies with marking of paths or community colorizing would make usability and readability very much higher, especially for learning or exploratory use.

In addition, although the package operates completely within INTERLISP, there is scope for developing a lightweight interface layer to integrate with modern tools (e.g., exporting output to external visualization, batch testing using scripting languages, or integration with Lisp-based web services).

This project also raises the potential for further exploration of modernizing legacy environments. INTERLISP LOOPS, for its age, still offers unique symbolic processing and object-oriented modeling facilities that remain relevant in domains like artificial intelligence, rule-based reasoning, or symbolic computation. The design patterns established in this package could serve as a blueprint for creating similar analytics frameworks in other domains — e.g.:

- Decision trees and symbolic classifiers implemented entirely in INTERLISP
- Knowledge representation systems centered on graph-based semantic models
- Symbolic optimizers or constraint solvers for educational or experimental use

Modularity and clarity of the package architecture will make it both a tool and a template — having utility for immediate practical work, but also pedagogical value for teaching system design using legacy computing platforms.

Lastly, this project has shown that it is still possible to achieve meaningful contributions in retrocomputing environments, and that with careful planning, even legacy systems like INTERLISP can be induced to host sophisticated algorithmic capability. The graph analysis package developed herein is not only a solution to a technical problem, but also an illustration of the way in which legacy environments can be extended, repurposed, and rendered relevant through careful engineering.

6 REFERENCES

[1] Kaisler, S. H. 1986. INTERLISP: The Language and Its Usage. Xerox Palo Alto Research Center.

[2] Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. Numer. Math. 1 (1959), 269–271.

[3] Bron, C., and Kerbosch, J. 1973. Algorithm 457: Finding all cliques of an undirected graph. Commun. ACM 16, 9 (Sept. 1973), 575–577.

[4] Newman, M. E. J. 2003. The structure and function of complex networks. SIAM Rev. 45, 2 (June 2003), 167–256.

[5] Hagberg, A. A., Schult, D. A., and Swart, P. J. 2008. Exploring network structure, dynamics, and function using NetworkX. In Proceedings of the 7th Python in Science Conference, 11–15.

[6] Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In Proc. Int. Conf. Learn. Represent. (ICLR ’17).

[7] Pfaff, T., Fortunato, M., Sanchez-Gonzalez, A., and Battaglia, P. W. 2020. Learning mesh-based simulation with graph networks. arXiv preprint arXiv:2010.03409.

[8] Cao, Y., Chai, M., Li, M., and Jiang, C. 2023. Bi-Stride Multi-Scale Graph Neural Network for Mesh-Based Physical Simulation. In Proc. Int. Conf. Learn. Represent. (ICLR ’23).

[9] Kaisler, S. H. 2024. MEDLEY-LOOPS: The Basic System.

[10] Kaisler, S. H. 2025. MEDLEY-LOOPS: Tools and Utilities. Addison-Wesley.

[11] Brown, J. 2005. A survey on legacy system modernization. IEEE Softw. 22, 4 (July 2005), 80–87.

[12] Watts, D. J., and Strogatz, S. H. 1998. Collective dynamics of ‘small-world’ networks. Nature 393, 6684 (June 1998), 440–442.

[13] Needham, M., and Hodler, F. 2019. Graph Algorithms. O’Reilly Media, Inc.