Methodology for Abstracting and Comparing Recipes using Directed Acyclic Graphs

By

Cole Tyler Peterson, Bachelor of Science in Computer Science

A thesis submitted to the Graduate Committee of

Ramapo College of New Jersey in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Fall, 2025

Committee Members:

Dr. Lawrence D'Antonio, Advisor

Dr. Amanda Beecher, Reader

Dr. Sourav Dutta, Reader

COPYRIGHT

DEDICATION

I would like to dedicate my thesis to my parents, Tom and Kathy, my brothers Kevin and David, and my nephew Thomas as well as all of those in my extended family (including friends), too numerous to name, but all equally dear to me. A very special dedication to my editor Janniry, my most trusted advisor and greatest resource for both mental fortitude and high-level writing experience.

ACKNOWLEDGEMENTS

I would like to give my thanks to the following people who supported me in my education as well as my thesis:

Dr. D'Antonio for his continued support and reassurance as well as teaching in both algorithm and Object-Oriented Programming design.

Dr. Beecher, for her teaching in data science, statistics, and graph theory. I greatly appreciate your advice, guidance, and enriching discussions.

Dr. Dutta, for his teaching of all things programming, especially in the Linux and Python ecosystems.

I would also like to thank Drs. Frees, Kowal, Kumar, McMurdy, Miller, Tweneboah, and Yuster from the School of Theoretical and Applied Science for an enlightening and unforgettable Ramapo experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
BET	Binary Expression Tree
DAG	Directed Acyclic Graph
GED	Graph Edit Distance
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
LAMP	Linux Apache MySQL PHP
LLM	Large Language Model
MVC	Model-View-Controller
OCR	Optical Character Recognition
OOP	Object-Oriented Programming

ABSTRACT

Recipes are a unique form of procedural knowledge that is historically lacking systematic analysis despite their relevance in culinary arts. While recipes appear to follow standardized formats, they rely heavily on implicit domain knowledge, cultural context, and unstated assumptions that make them difficult to parse, compare, and quantitatively analyze. No publicly available tools exist to decompose recipes into discrete components that would enable programmatic analysis, comparison across recipe variations, or extraction of underlying cooking principles. This gap prevents systematic study of culinary knowledge and development of tools that could make recipes more accessible to users of varying cooking backgrounds and skills. This research addresses these limitations by developing a novel methodology for representing recipes as directed acyclic graphs (DAGs), partially inspired by binary expression trees (BET) where ingredients and equipment used in a recipe represent operands and cooking actions represent operators. We had three main objectives: (1) create a framework based on existing data structures and programming tools capable of abstracting any recipe regardless of complexity, semantics, or cultural context. (2) Identify quantitative metrics that could be extracted from recipe DAGs to calculate recipe complexity and enable meaningful comparisons between them. And (3) to develop a software toolset for recipe ingestion, manipulation, visualization, and analysis through a standardized API. Our analysis revealed significant complexity and variation within the dataset, with critical path lengths ranging from 4 to 48 edges (average 17.1). The dataset contained 203 unique ingredients across 815 occurrences, with high specialization (136 ingredients appearing in only single recipes). Dominant ingredients include sugar (64.5% of recipes), flour (55.3%), and eggs (52.6%). Co-occurrence analysis revealed fundamental baking patterns, with flour + sugar appearing in 46.1% of recipes. Intermediate product comparison

using Jaccard similarity (mean: 0.052) demonstrated that intermediate naming conventions capture functional roles rather than strict compositional specifications. The weighted complexity formula $C = 0.5B + 0.35A + 0.15I$ enabled skill-level classification into beginner, advanced, and expert categories based on branch count, action count, and ingredient count. This work establishes a comprehensive methodology for systematic recipe analysis through graph-theoric approaches, providing a foundation for future applications in recipe recommendation systems, instruction generation, cooking education, and temporal/cultural culinary analysis.

Organization of the Thesis

The thesis is structured in the following manner:

Chapter 1 Provides relevant background information on our research topic, introduces our research problem, outlines the research questions and objectives, and presents the novel contributions of this thesis topic.

Chapter 2 Provides a literature review that identifies relevant works while detailing how the present research contributes to the body of literature and understanding on the topic.

Chapter 3 Details the research methodology including data collection, manipulation, and analyses run to compare directed acyclic graphs. As well as giving a detailed description of the programming work.

Chapter 4 Presents the results from the analyses of the work.

Chapter 5 Concludes the thesis by summarizing key contributions and offers recommendations for future work.

CHAPTER ONE: INTRODUCTION

1.1 Overview

Recipes appear simple at first glance but to those with little cooking skill they can be riddles highly influenced by tradition and assumptions present in the culinary field. Understanding recipes and creating them requires domain-specific knowledge unique to cooking that culinary professionals not only understand but consider to be "unspoken rules" (Mori et al., 2014). Within these unspoken rules and overall language preserved in recipes is knowledge from experience, intuition and domain-specific cooking knowledge. Take for example combining water, flour, and salt, to a professional they are immediately able to identify that these ingredients give us a rudimentary dough, someone who is not well-versed in cooking might not know this. Because of this recipes are a good study subject to understand the challenges of formalizing tacit knowledge into explicit, analyzable structures that can bridge expertise gaps.

Since recipes are built on unstated rules and missing structure, we lack a system for abstracting, categorizing, or comparing them; no publicly available tools decompose recipes into ingredients, equipment, and actions in a way that allows for programmatic analysis (Donatelli et al., 2021). Our work aims to fill that gap by creating a methodology that reconstructs recipes from the bottom up using parts of speech, semantics, and explicit rule-codification. With recipes decomposed into common components, tasks like serving-size conversion, set comparison with ingredient inventories (e.g. home pantry or commercial stock), or enrichment with metadata such as nutrition or cost become trivial. This abstraction would also make it possible to apply common graph theory approaches, track how recipes evolve across time, cultures, personal preferences,

and highlight cooking principles that are usually buried in tacit knowledge. Even the most popular recipe blogs and repositories are static, and are assessed by dedicated editors to ensure their accuracy, repeatability, and originality (AllRecipes, 2024).

To address this we will develop methodology that will allow us to find insights and cooking tips (often information not explicitly stated) in a recipe by utilizing directed acyclic graphs (DAGs). DAGS are graphs that represent dependencies through directed edges, where nodes connect in a specific direction and no node can be visited more than once. This makes them ideal for representing sequential processes such as recipes. They can tell us the complexity of recipes through metrics such as the number of transformations ingredients undergo, as well as relationships between recipe components. To get these metrics we can look at the following aspects of DAGs: path lengths (including critical/maximal paths and average path lengths), branch counts, and topographical ordering which maps prerequisite actions and ingredient combinations.

Beyond recipes, this work has broader implications for any domain involving transformative processes with clearly defined inputs and outputs. For instance, in manufacturing, the same DAG methodology could represent production workflows where raw materials serve as base inputs, machinery and workspaces function as equipment nodes, and discrete manufacturing operations act as transformation actions that incrementally convert materials into finished products. The ability to formalize, compare, and optimize such processes through graph-theoric analysis could enable applications in supply chain management, production scheduling, and quality control across industrial settings.

1.2 Problem Statement

Recipes are commonly found in online recipe blogs, published cookbooks, or familial recipe cards. When it comes to familial recipe cards, they are often written by someone of an older generation, creating contextual information that may not be universally understood. For example the recipe cards we used for our dataset were written by a nurse, who has worked in the industry since well before computerized records. Many of the recipes use the symbol C̄, in medical abbreviation, this means "with". The issue with this is that without the context relating to the recipe writer's background, a reader might not be able to parse these symbols into clear steps.

For recipe blogs, many users find the extensive personal narratives preceding recipes to be a barrier in quickly accessing the ingredients list and instructions for a recipe (Bryan, 2019). There are two main factors behind this verbosity: search engine optimization (SEO) and copyright protection. Longer content with personal stories and keyword-rich text tends to rank higher in search results, incentivizing bloggers to include extensive preambles before the actual recipe (Bryan, 2019). Additionally, the personal elements allow authors to claim their work as original creative content protected under copyright law; the combination of ingredients and required actions themselves are not protected, but the language describing the recipe or its background are (U.S. Copyright Office, 2023). The majority of popular recipe websites therefore contain large blog posts with pictures, explanations, and personal experiences embedded alongside recipe metadata in a more standardized form (Schema.org, 2024). For average readers, this structure makes it difficult to extract supplemental cooking information from the personalized writing. For experienced readers or those seeking recipes for comparison

or reference, they must scroll through the majority of the page before arriving at the recipe in a concise and recognizable form.

Our system has the potential to parse any recipe and ignore qualifiers or parts of speech that are not considered necessary to follow the steps and create the product out of its constituent components. As there are numerous ways to refer to the same ingredients (e.g. different forms/styles of the same ingredient, colloquialisms, brand names), the system would have to track the relationship between an ingredient and the numerous forms that refer to it. This should mitigate any barriers related to perceived cooking skill that one may encounter in any recipe, and provide transparent information about how complexity is determined in an analytical way.

1.3 Research Questions

1. Is there an existing data structure that can allow for comparison of elements that contain a wide range of associated metadata?
2. What can quantitative metrics of DAGs such as maximal (critical) and average path lengths from ingredient nodes to the final product actually mean in the context of cooking?

1.4 Research Objectives

To address our first research question, we will identify or develop a data structure capable of storing deconstructed recipes along with their associated metadata (ingredients, actions, equipment, duration, and symbolic aliases). We will evaluate DAGs as a candidate structure

based on existing graph theory methods with available algorithms for composition, traversal, and comparison. If existing structures are insufficient, we will adapt one that can represent recipe components and their dependencies while preserving associated metadata. For our second research question, we will define and implement quantitative metrics derived from recipe DAG structures: critical path length, average path length, and branch count, as well as interpret their meaning in the context of recipe complexity and cooking skill recommendation. These metrics will enable meaningful comparison across recipes that vary widely in ingredient diversity, proportions, and procedural depth.

CHAPTER TWO: BACKGROUND AND LITERATURE REVIEW

2.1 Introduction

In order to answer our research questions, we have to understand how recipes can be abstracted, as well as how semantics, difficulty, and context can be extrapolated through quantitative metrics in a DAG. We would also need to look for established methodology used for comparing DAGs based on these metrics. These goals facilitated a literature of DAG comparison through several methods, technical writing about recipes is likely much more obscure and difficult to source. One of our main goals was accessing what mathematical tools were available for analysis of the composition of recipes. We searched for literature via Google Scholar and Ramapo library resources. The topic being obscure and abstract yielded only 12 papers, some manuscripts or theses, but all still relevant to the overall topic. All papers are in the reference section but only certain ones are cited within text when discussing aspects of the work that are relevant to our goals.

2.2 Literature Reviewed

The current literature has some foundation for the DAG comparison and understanding the complexity of different graphs. For example, there is much information on graph edit distance (GED). GED is a metric that defines distance as the minimum-cost sequence of node and edge insertions, deletions, and substitutions to transform one graph into another (Gao et al., 2010). It is widely used in pattern recognition (images, handwriting, molecules, fingerprints, etc.)

wherever relational structure matters (Algabli, 2020). For recipe DAGs, GED could provide methodology to calculate how "far apart" two recipes are in terms of structure and operations. However, this method has drawbacks, GED is computationally expensive for general graphs, which restricts scalability as dataset size increases (Gao et al., 2010). For recipe comparison, the hand-tuned edit costs would need to reflect culinary semantics (e.g. substituting butter for margarine should cost less than substituting flour for eggs), which requires domain expertise to calibrate and may vary through many different culinary/dietary contexts.

Measuring and comparing DAG similarity and taxonomy are also detailed in the literature, specifically DAG distance based on partial orders (Malmi et al., n.d.). Some papers extend Kendall-tau distance from total orders to DAGs with a shared vertex set, penalizing discordant, concordant, and incomparable pairs with tunable parameters. Their distance satisfies a relaxed triangle inequality and supports DAG aggregation and clustering, albeit with NP-hard optimization problems that account for approximations (Malmi et al., n.d.). This approach assumes there is a shared vertex set between compared DAGs, which recipes rarely have; different recipes use different ingredient sets, making direct node correspondence undefined. The only exception to this would be recipes of the same specific dish such as the 2 nearly identical cheesecake recipes in our dataset.

One paper in particular proposes a Katz-based similarity measure (Nayak et al., n.d.) for comparing large taxonomies modeled as DAGs, focusing on knowledge hierarchy evolution (e.g., DBpedia; a Wikipedia structured data project). Their measure leverages weighted path counts to capture both structural similarity and logical subsumption, provides a linear-time variant, and is tunable for different applications. This method explicitly highlights limitations of prior DAG/taxonomy similarity measures (e.g., tree based, edge overlap metrics, or Fowlkes

Mallows) in terms of scalability and failure to capture semantic categorization. While effective for taxonomic hierarchies with consistent parent-child relationships, recipes exhibit more complex dependency patterns including parallel branches, optional steps, and ingredient substitutions that this measure does not accommodate for.

DAG-Coding is also present in the literature, this method converts continuous temporal streams (e.g. handwriting trajectories) into DAGs through segmentation and multi-path representation (Lin & Kung, n.d.) Similarity between streams is computed as a path-matching score on DAGs using a "DAG Compare" algorithm, which supports multi-path segmentations and is resilient against segmentation errors. This is a conceptual precedent for turning recipes as timelines of actions into DAGs, especially when there is ambiguity in segmentation (merged or split steps, overlapping actions). This methodology is ultimately more suited for continuous trajectories like handwriting, not for discrete procedural steps with explicit dependencies

Domain-specific DAG comparison via edit scripts (Eder & Wiggisser, 2006) compare versions of data-warehouse dimension hierarchies (rooted, ordered DAGs) using a script with operations such as node insert/delete, edge insert/delete, value update, and renaming. It was designed for small structural differences between successive versions, making it conceptually close to recipe "revision" scenarios (e.g., adjusting timing, inserting an extra step, substituting ingredients). Tree edit distance as a restricted baseline (Demaine et al., 2006) provides a cubic-time algorithm for tree edit distance through several decomposition strategies. This could be relevant as the recipe structure can be approximated as a tree (no shared substeps), allowing for a computationally cheaper baseline to contrast with full DAG comparison methods. However, recipes frequently contain shared intermediate products (e.g. a batter used in multiple steps) that violate tree assumption, making full DAG comparison necessary. These edit scripts also assume

hierarchical dimension structures rather than the branching and merging patterns common in recipes that contain sub-recipes that represent processes to be performed concurrently

Most relevant to our work, Donatelli et al. (2021) develops a methodology for aligning actions across recipe graphs to enable cross-lingual recipe understanding. Their approach constructs flow graphs from recipe texts where nodes similarly represent ingredients, action, and intermediate products connected by directed edges indicating information flow. The work demonstrates that recipe graphs can be aligned across different languages of the same recipe using graph matching algorithms. Our methodology both differs from and extends this work in several ways. While Donatelli et al. (2021) focuses on action alignment for cross-lingual mapping, our work emphasizes quantitative complexity analysis through metrics such as critical path lengths, branch counts, and weighted complexity scores that allow for skill-level classification. Their flow graphs also do not incorporate producer-consumer semantics, where actions explicitly consume ingredients and produce intermediate products with inherited properties. Next, our system includes canonical symbol mapping that normalizes ingredient aliases, enabling comparison across recipes that vary in terminology. Finally, the paper does not provide anything past a theoretical graph representation, we have created practical tools for the analysis of recipes.

2.3 Conclusion

Our work aims to fill the knowledge gap by capturing recipe semantics and human cooking behavior explicitly; previous work neither quantifies the cognitive load of managing concurrent cooking processes nor provides practical software tools for recipe decomposition and complexity comparison. None of the literature acknowledges the real-world implications of performing processes in physical space, only in a mathematical or theoretical sense. Compared to the mentioned papers, our methodology should allow for novel ways to:

1. Represent recipes as DAGs in a way that respects culinary semantics.
2. Contextualize edit costs from human relevant signals (difficulty ratings, complexity scores, etc.)
3. Interpret DAG similarity as a measure of recipe complexity or cooking skill.

CHAPTER THREE: METHODOLOGY

3.1 Introduction

A Directed Acyclic Graph (DAG) is a type of graph where nodes connect through edges that represent directionality. No node can be visited more than once, making them useful for representing dependencies between nodes such as in scheduling tasks. One example of where DAGs are utilized deals with students scheduling courses in school; if class A and class B are prerequisites of class C, class C can never be taken until both A and B are completed. This is similar to process scheduling in Control Processing Units (CPUs), as whatever order in which they are scheduled will not violate the fact that prerequisite processes must be performed before subsequent ones. Let's say we have 6 arbitrary tasks labeled A-F, each with their own time to completion (Figure 3.1).

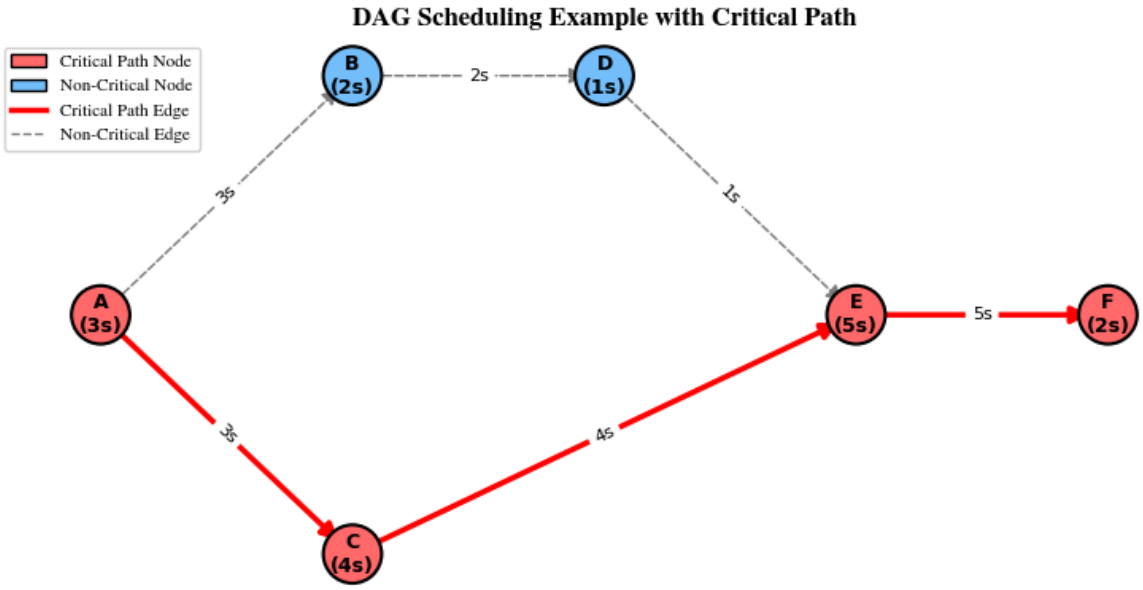


Figure 3.1 - DAG Scheduling Example with Critical Path

We can see that there are 2 paths in which we can complete these tasks sequentially:

Path 1: $A \rightarrow B \rightarrow D \rightarrow E \rightarrow F = 3s + 2s + 1s + 5s + 2s = 13s$

Path 2: $A \rightarrow C \rightarrow E \rightarrow F = 3s + 4s + 5s + 2s = 14s$

We would consider path 2 the "critical" path as it is the longest time that any task sequence can take. This time represents the minimum completion time for the whole scheduling process.

The choice to use a DAG to represent recipes was partially inspired by Binary Expression Trees (BETs), with operators representing actions that transform operands which represent ingredients and equipment required, ultimately resulting in a final product (Figure 3.2). This in theory should allow a recipe to be represented by large arithmetic operations, leveraging the core concepts of DAGs to preserve topographical ordering. In order to quantify the complexity of our Direct Acyclic Graphs we will gather a set of metrics (detailed in the next section) that we can use to calculate complexity of recipes relative to each other.

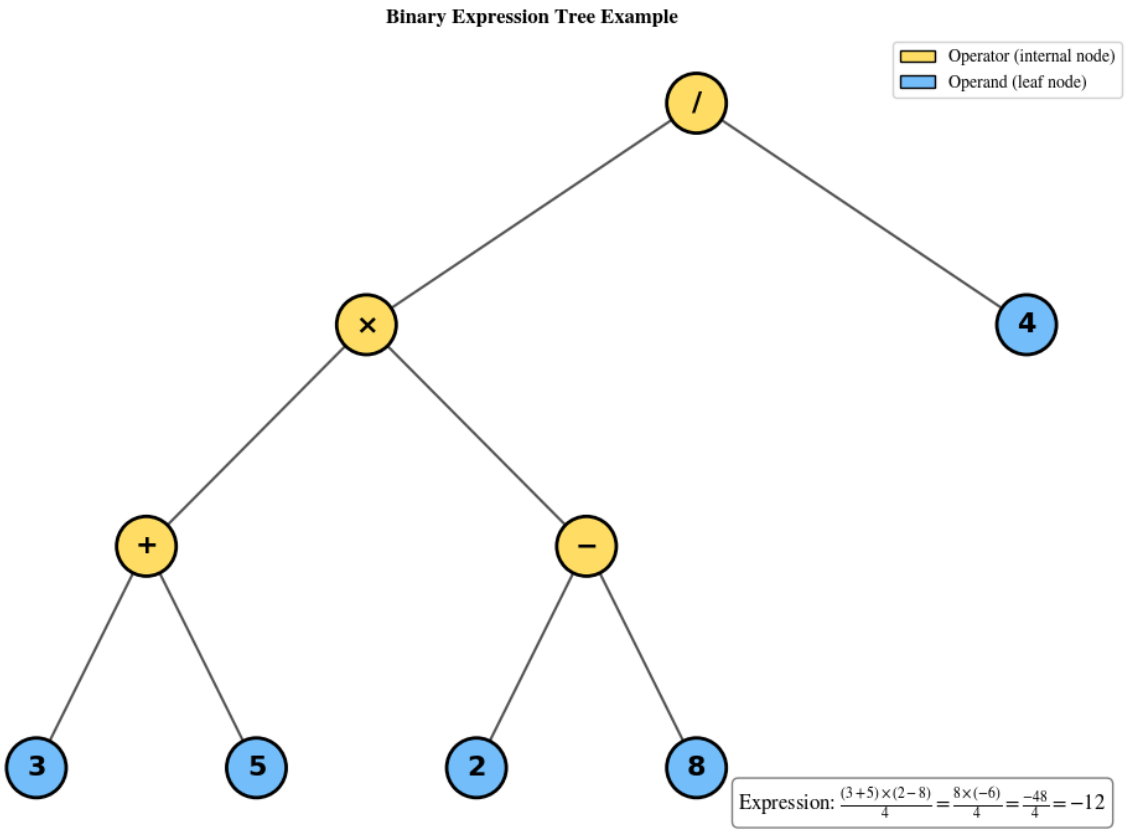


Figure 3.2 - Binary Expression Tree Example

3.2 The Directed Acyclic Graph structure, key metrics and variables

While prior work has constructed graphs from recipe text (Donatelli et al., 2021; Mori et al., 2014), no existing algorithms incorporate the combination of producer-consumer semantics, canonical symbol mapping, and quantitative complexity metrics that our methodology provides. Additionally, the work also has practical significance, because there exists no programming toolsets for the ingest, manipulation, visualization, and comparison of recipes; all existing literature is theoretical. Access to these tools will be controlled through an Application Programming Interface (API). The API will expose all of our tools and data structures to a

frontend application such as a browser-based Graphical User Interface (GUI) in a standardized way.

Directed Acyclic Graphs have the following structure:

(1) Directed - Meaning that the edges of the graph traverse in a specific direction.

(2) Acyclic - Meaning traversing the directed edges through the entire graph, no previously visited nodes are revisited; no paths form a cycle. In the case of recipes, no previous actions can be returned to as they have already transformed ingredients irreversibly into intermediate products.

(3) Adjacency list - A structure that maps every node in a directed graph to a list of all nodes that it points to, allowing for immediate lookup of any given node's successors or predecessors. This allows for a space-efficient method of checking what prerequisite actions must be performed and with what combination of ingredients before any given action.

(4) Topographical ordering(s) - for every directed edge u → v, u comes before v (Cormen et al., 2009), topographical orderings are non-unique as there may be multiple ways to achieve a final product without violating the DAG rules, for example the production of both a cake and icing can technically happen in either order, as long as they are both consumed to form the final product.

<u>The key metrics and variables of DAGs that will be considered in this data set:</u>

(1) Branch count: the number of branches that produce intermediate products. These are usually joined back into the main branch when combined with another intermediate product. Branches may be indicative of complexity as they can represent sub-recipes (icing/topping), that are performed while long parallelizable actions (baking or otherwise

cooking) are occurring in the main branch (Figure 3.3). We are defining a branch as an independent preparation path in a recipe that originates from a root instruction, having no prerequisite inputs from other instructions. Branch convergence occurs when an action consumes outputs from multiple branches, creating a synchronization point where all prerequisite branches must complete before proceeding. The branch count B equals the number of root instructions in the recipe; recipes with $B = 1$ are sequential, while $B > 1$ indicates concurrent preparation requiring multi-tasking in a recipe.

(2) Path lengths from ingredients to final products: iterate through the adjacency list and count the edges between nodes only in the paths from ingredients (leaves) to the final product (root) of the graph. We then store lengths as values in a dictionary, with their unique key being the path (e.g. path A→B→C, has a length of 2 edges).

(3) Critical path length(s): using our ingredients→final product paths subset, we sort by ascending path length, and those with the maximum number of edges are our critical paths. Requires topographical ordering, in which the nodes are ordered linearly in a way that guarantees all prerequisite nodes have been traversed before the current node. The critical paths represent the ingredients that must undergo the most amount of transformations before the final product is achieved (Figure 3.3). Large critical path lengths might be indicative of high complexity in a recipe.

(4) Average path length(s): this also uses the ingredients→final product paths subset, we also sort by ascending path length and take the average of all of these edge counts. In terms of the recipe this would tell us the average number of transformations ingredients must undergo through actions in order to produce the final product.

(5) Weighted average of recipe component counts: we can create a basic formula based on the weighted average of recipe components such as concurrent process count (branches), unique action count, and ingredient count. These metrics are all related to the overall complexity of the recipe; they cover multitasking operations, as well as recipe breadth and depth, respectively

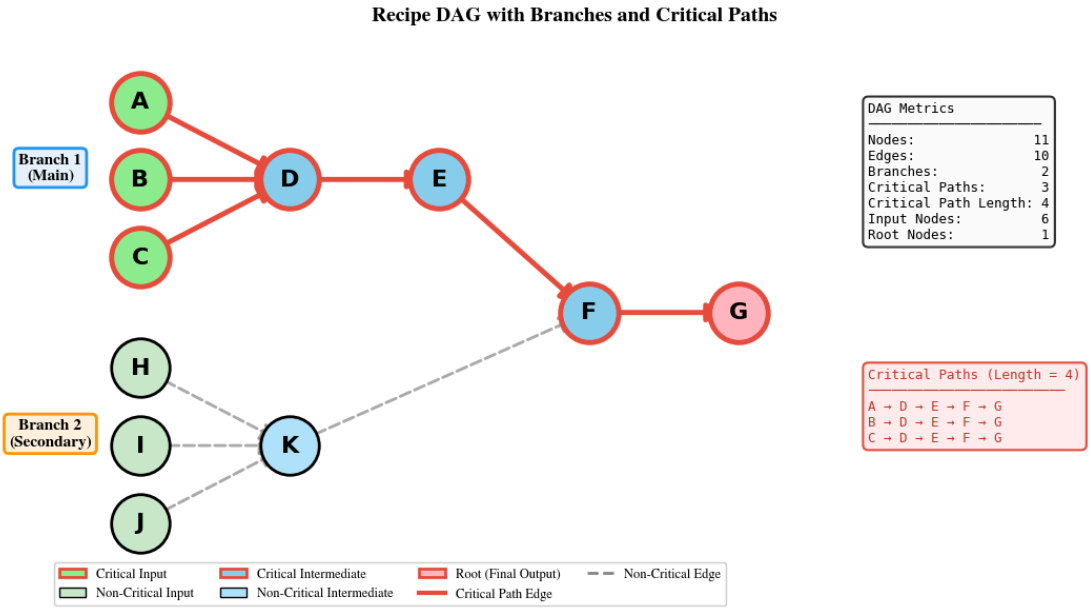


Figure 3.3 - Recipe DAG with Branches and Critical Paths

3.3 Data Collection, Cleaning, and Manipulation

To gather data for the dataset, I utilized familial recipes and subsetted the recipes that used “baking” methods and skills to produce their final products (Appendix A). This would reduce the variation of symbols for comparison between recipes. The recipes were on handwritten recipe cards (Figure 3.4), many had non-implicit instructions and various shorthand symbols. These recipes were translated into plain text files, revised for spelling and grammar,

then clarified with the owner of the recipes regarding spelling, handwriting, and shorthand symbols.

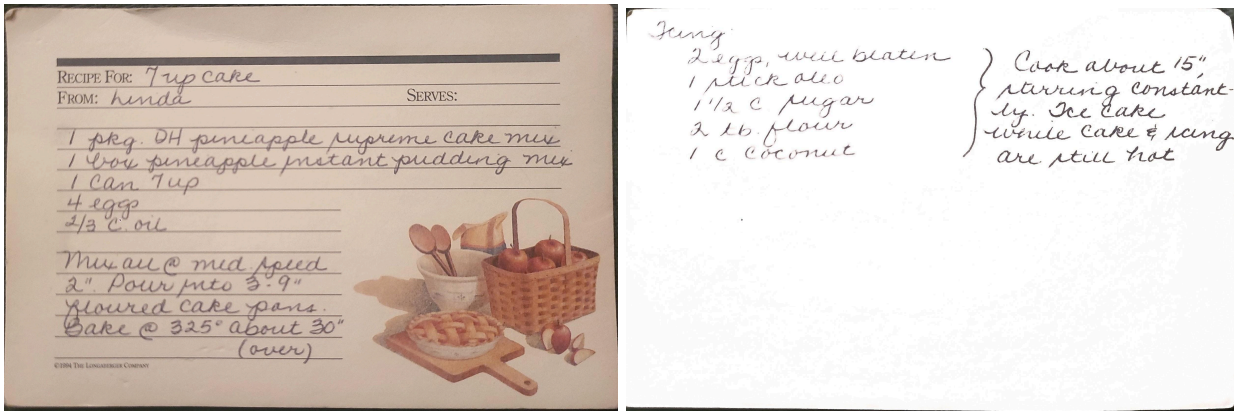


Figure 3.4 - 7 Up Cake Recipe Card (Obverse and Reverse)

Following this, I began to build the schema, breaking apart the text files by parts of speech, translating them into structures in a JavaScript Object Notation (JSON) schema (e.g. ingredients, actions, equipment, durations, units, properties, delimiters) representing a source of truth for the storage of recipes and all of their constituent components. The JSON recipe schema has properties of linked-lists and stores all contextual information about each instruction in the form of ingredients and equipment used, the time that action requires to finish (bake time), and any properties or other states that must be achieved by intermediate products for completion of the task. (e.g beat until stiff peaks for icing). All terms in a recipe that represent a component of it are symbols. Symbols include the physical items that are the inputs into each action (ingredients and equipment), as well as the actions that process these inputs and produce intermediate products. Units of measurement such as time, mass, volume (of dry and liquid ingredients), and temperature are also considered symbols (Figure 3.5). If any items or actions are combined, their identities and properties are combined using a rule lookup table into intermediate forms, which are considered derivative of their parent symbol types.

I then wrote a Python script to scrape every recipe JSON file by symbol, building a relational table for all symbols of every type. Each table represents symbols within a certain type that may describe the same real-world ingredient, so they are essentially tables of ingredient aliases. A new table was created to store "canonical" forms, then all of the aliases were assessed to map them to a canonical symbol that was representative of all of its aliases. For example, “white sugar”, “granulated sugar”, and “sugar” in recipe ingredient lists are generally understood to describe the same ingredient so we codified the following mapping: (granulated sugar, sugar, white sugar → granulated white sugar). These aliases allow us to track different forms and terms that ingredients are known by, mapping them to canonical forms which we use internally to ensure no ambiguity between symbols.

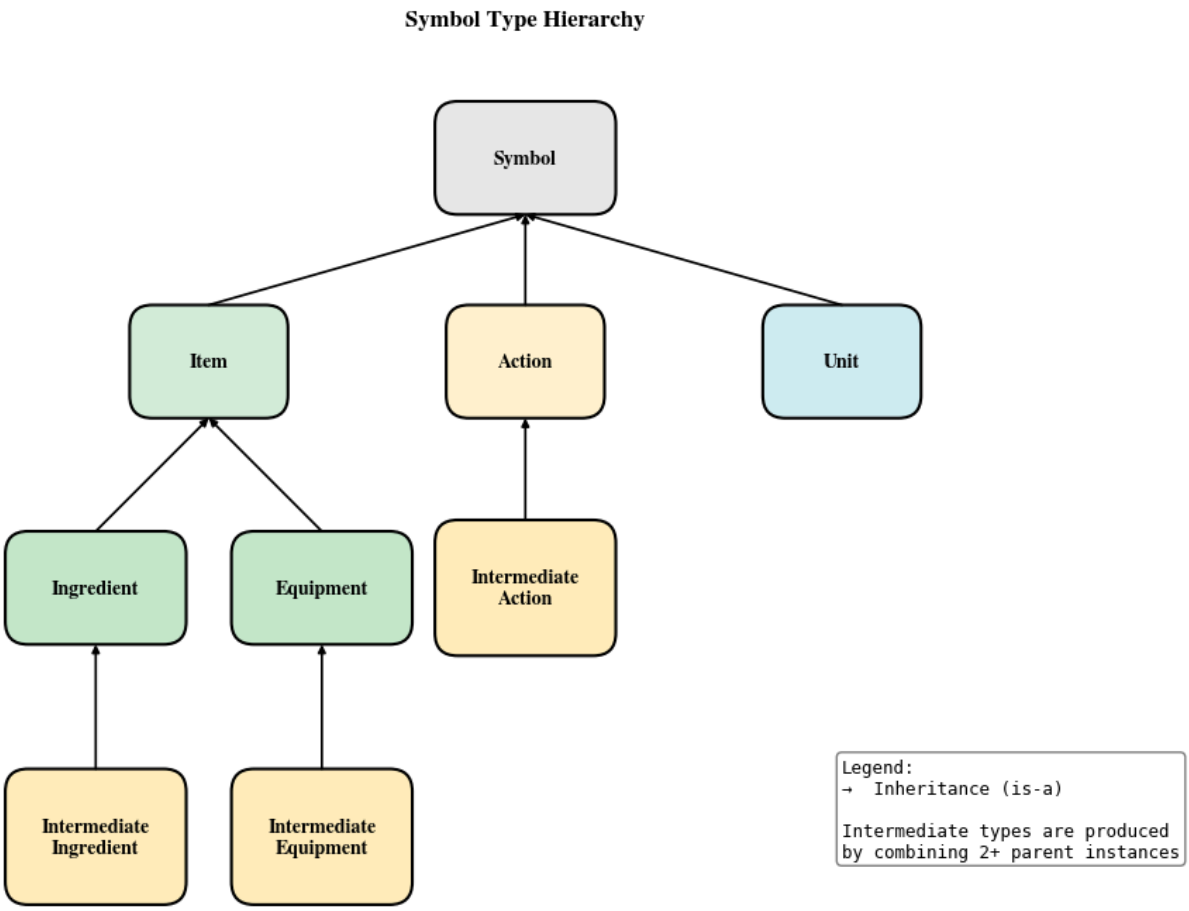


Figure 3.5 - Symbol Type Hierarchy

3.4 Program Structure and Toolset Development

Before going into detail about our various programming layers, we must discuss how we actually build DAGs out of recipes. Our DAG building algorithm transforms a recipe's ingredient and instruction sets into a directed acyclic graph that represents a procedural execution of a recipe (Appendix D). Given recipe steps $S = S_1, S_2, ..., S_n$ and ingredients $I_1, I_2, ..., I_n$, the algorithm produces a graph $G(V, E)$ where nodes represent ingredients,

equipment, actions, and intermediate products, while edges encode prerequisite and producer-consumer relationships. The algorithm operates in four passes:

1. Initialization - tracks equipment and ingredient states, vessel contents, and producer-consumer mappings
2. Branch Assignment - identifies root instructions (nodes with no prerequisites) as separate preparation paths and assigns branch identifiers through topological ordering
3. Level Computation - assigns temporal levels to each instruction using a multi-pass approach: first computing natural levels based on prerequisites where outputs appear at CompletionLevel + 1 and consumers execute at OutputLevel + 1, then adjusting concurrent instruction groups (identified via transitive closure of concurrentWith action relationships) to share the same level, then enforcing cross-branch sequential execution so that only one non-concurrent action occupies each level, then finally extending completion levels for duration-based actions to span multiple levels while parallel branches execute
4. Graph Construction - create nodes for each ingredient and equipment usage, action nodes at their assigned start levels, and output nodes at completion levels plus one, connecting them via prerequisite edges (for temporal dependencies) and consumption edges (for data flow)

Our main algorithm also employs several helper functions such as: NormalizeAndCompare, which converts measurements into standard units for proportion calculations. CombineActions merges concurrent actions into single intermediate nodes by joining names and unifying properties and identities. CombineIngredients and ProcessIngredients

handle ingredient combination and transformation based on action semantics (e.g. “fry” adds “fried” prefix and updates the intermediate product identity). ProcessEquipment applies action-based transformations to equipment states (e.g. heating a cooking vessel), HandleVesselContents manages vessel content tracking when ingredients are added to or removed from vessel equipment. The resulting DAG supports multiple independent branches that execute sequentially unless actions are explicitly marked as concurrent, with branch convergence occurring when an instruction consumes the intermediate products from multiple branches, waiting for all prerequisites to complete before proceeding

For decomposing recipes into abstract symbols, running analyses on them, and viewing detailed information about their structure, we have defined several “layers” that various programming tools will operate within to achieve these goals. This layer-based approach is important, as the base tools that are used across the entire project can be interfaced with in the form of Python shell commands, as importable Python modules, or through the API. All layers and the software that compose them are defined through Docker Compose files, stored in both the front and back end code repositories. Docker allows for the definition of all software required to represent these layers, as well as all environmental information and configuration required to automate the orchestration of the software in containers. These containers are analogous to virtual machines, where any user with Docker installed and our project files can recreate the exact same environment that we are using to develop and stage our project for consistency.

The most base layer is that of storage, the data that describes each recipe can be separated into the recipes themselves, and the metadata that is supplementally used to describe all of the symbols within a recipe. For storing the recipes themselves we are using MongoDB which holds large JSON documents (in this case, recipes stored in our internal recipe schema). The metadata

for the symbols referenced in these JSON documents is stored in MariaDB, a relational database that uses Structured Query Language (SQL) for efficient retrieval and union of data. After gathering all of the recipes as JSON documents represented by their base symbols and properties, we designed a relational schema to hold all symbol types and instances within the MariaDB database (see Appendix B) With these two database softwares, we can ensure that recipes and symbol metadata are only created/retrieved/updated/deleted in MongoDB through the appropriate application-layer functions. This strategy aims to limit the areas of code that produce side-effects, ensuring that the areas of the application layer that manipulate this data only perform one type of task each.

The application layer is composed of Python modules, employing a Model-View-Controller (MVC) architecture. This architecture uses Object-Oriented Programming (OOP) principles to define models, which represent elements in our OOP class structure (see Appendix C). The only function of models in this context is to be populated with data retrieved through from the object and relational databases. This allows us to have well-defined components that represent all aspects of our recipe structure, ensuring that they can be handled together using common interfaces. These interfaces take the form of controllers, classes that handle the population of the models with database information. Controllers are also responsible for the safe manipulation of recipes as well as passing off relevant model data to various utility modules including recipe visualization and statistical calculation. For visualizations of recipe DAGs, we utilized the Python package NetworkX that contains various data structures and common algorithms for manipulating graphs as well as GraphViz, a graphics package. GraphViz was not as customizable as we liked (frequent formatting issues and the inability to sort input nodes), so it was replaced with the Matplotlib, another data visualization

package. This allowed us to then leverage the mpld3 package which converted our Matplotlib graphs into interactive HTML elements for use in the frontend. The final leg of the MVC architecture is views, which are the specific contexts that the controller passes model information to. Views can include the display of only statistical information or contexts that include either a subset of or all recipes and various visibilities of their structural components.

In our case, the controllers pass this contextual information through the API for use in our frontend application. The API is based on the Python FastAPI framework which provides the easy definition of API endpoints that are served over the Hypertext Transfer Protocol (HTTP) for use in web applications. These endpoints expose functions of our Python controllers in a controlled way (user access control, rate-limiting, error handling) so that there are no layer violations; the frontend can only use API endpoints which carefully handle the manipulation of data through the controllers.

The frontend software is defined in Docker as a classic Linux Apache MySQL PHP (LAMP) stack. The containers use lightweight Alpine Linux images to run the Apache HTTP server, which hosts web pages and facilitates the serving of web resources to users' browser applications. PHP is the interpretive "scripting" language that allows for the implementation of web frameworks used in the facilitation of data between the API and frontend layers. In our frontend, we are using the PHP framework Laravel which has a similar MVC architecture to that of our backend Python code. Laravel allows for the rapid development of routes for either internal, administrative, or public use. For example, only users authorized in Laravel as "administrators" can access functions that contain information that should not be visible to the public such as system status and privileged user information. Laravel also serves reactive graphical components through Vue.js, a Node.js/JavaScript framework for defining elements that

represent aspects of our recipe models, as well as common components for interacting with them. This setup allows for a robust way to develop the web server functions themselves as well as components that leverage common data sources and graphical elements for speed. Laravel uses a SQL database to store frontend application information, we are using the same MariaDB instance that our backend uses to store symbol metadata. This structure also helps to further retain layer-separation, so that core backend functions, API endpoints, web server functions, and reusable graphical components can be developed and tested independently.

3.5 Recipe Complexity and Comparison Criteria

For understanding how recipe DAGs can be grouped based on complexity we can define basic criteria such as counts of the following DAG metrics: branches, unique actions, and unique ingredients. We can then assign weights based on our perceived importance of the criteria and use the following formula to calculate recipe complexity C:

$$C = w_b \cdot B + w_a \cdot A + w_i \cdot I$$

Where:

- B = Branch Count (concurrent processes to manage)
- A = Unique Action Count (recipe depth)
- I = Unique Ingredient Count (recipe breadth)

Weights (sum 1.0):

- $w_b = 0.5$
- $w_a = 0.35$
- $w_i = 0.15$

We decided to weigh branch count as 50% of a recipe's complexity, as branches represent concurrent processes that are performed alongside the main ones (e.g. while a cake is baking, cook an icing). The skill required to manage multiple cooking processes at once should indicate higher overall cooking skill. Unique action counts comprise 35% of complexity scoring; action variety is not as important as concurrent processes management, but it is more important than ingredient variety. As action variety increases, so does the overall depth (number of levels) of the recipe DAG. We consider ingredient count to be the lowest complexity factor, as large amounts of ingredients do not necessarily equate to more actions being performed in a recipe. This weight is justified through recipes such as Apple Cake which has 10 ingredients that are all consumed by a single combining "mix" action (Figure 3.6). Using these weights, we can take the counts for each recipe and solve for C.

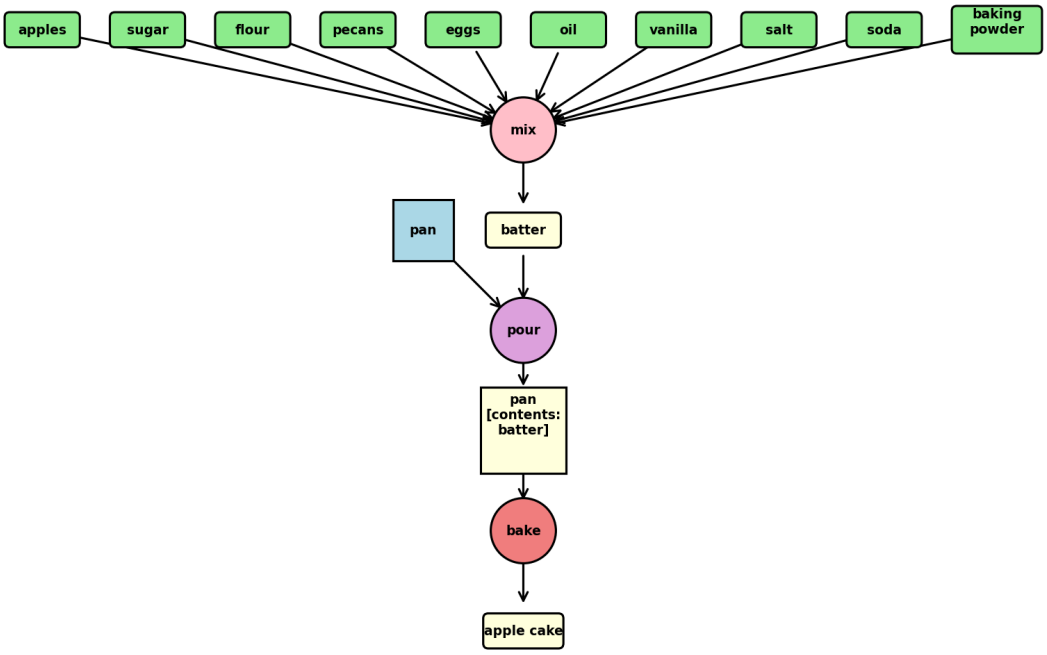


Figure 3.6 - Apple Cake DAG Visualization

Skill Level Classification:

After calculating complexity scores for all recipes using the weighted formula, we classify recipes into skill levels using percentile-based thresholds. The 50th percentile (P_{50}) serves as the boundary between beginner and advanced recipes, representing the median complexity of the dataset. The 85th percentile (P_{85}) denotes advanced from expert recipes, ensuring that only the most complex 15% of recipes receive the expert classification.

Skill Level	**Complexity Score Range**	**Percentile**
Beginner	$C < P_{50}$	Below 50th percentile
Advanced	$P_{50} \leq C < P_{85}$	50th to 85th percentile
Expert	$C \geq P_{85}$	Above 85th percentile

Table 1 - Skill Level Classification Thresholds

This percentile-based approach has two main advantages over fixed threshold values. Firstly, it adapts to the complexity distribution of any dataset, ensuring meaningful stratification regardless of absolute score ranges. Secondly, it also produces interpretable skill categories where “beginner” recipes are easier than half of all recipes in the dataset, “advanced” recipes are harder than most but not exceptionally difficult, and “expert” recipes represent the highest tier of complexity in the dataset. The percentile values 50 and 85 were chosen to create roughly balanced beginner/advanced groups while reserving expert status for recipes that require exceptional multitasking and procedural depth.

3.6 Directed Acyclic Graph Structure Analysis Methodology

Path length statistics quantify the transformational depth of recipes. For each recipe DAG, we enumerate all directed paths from ingredients nodes to the final product node. Each path's length is measured in edges, where one edge represents one transformation step. Three metrics are computed per recipe: the minimum path length is the shortest ingredient transformation chain in a recipe, often representing ingredients such as garnishes or toppings added near completion. A good example of a recipe that would have many minimum paths is that of a salad, all ingredients require minimal transformation before they are simply combined together, representing a high ratio of ingredient inputs to transformations. The maximum path length or critical path length indicates longest ingredient transformation chains, representing the most extensively processed ingredients. This path metric is the limiting factor for recipe complexity; the longest path in the recipe represents the minimum steps to completion. The last metric is the mean path length which is the average transformation depth of ingredients across the whole recipe. When reporting dataset-wide statistics, we aggregate these per-recipe values: "average minimum path" is the mean of all recipes' minimum paths, "average mean path" is the mean of all recipes' mean paths, and "average critical path" is the mean of all recipes' maximum paths. The range reports the minimum and maximum critical path lengths observed across all recipes. For example, consider a recipe where flour undergoes 8 transformations (mix→knead→rise→punch→shape→proof→bake→cool) while a glaze is simply mixed and applied (2 transformations). This recipe would have a minimum path length of 2 edges, critical path length of 8 edges, and a mean path length of 5 edges.

Metric	Per-Recipe Formula	Aggregated Formula
Minimum path length	$min\left(\left\{\lvert p\rvert : p \in paths_r\right\}\right)$	$\frac{1}{\lvert R\rvert}\sum_{r\in R} min_r$
Mean Path Length	$\frac{1}{\lvert paths_r\rvert}\sum_{p\in paths_r}\lvert p\rvert$	$\frac{1}{\lvert R\rvert}\sum_{r\in R} mean_r$
Maximum (Critical) Path Length	$max\left(\left\{\lvert p\rvert : p \in paths_r\right\}\right)$	$\frac{1}{\lvert R\rvert}\sum_{r\in R} max_r$

Table 2 - Path Length Metric Formulas

Where:

- $paths_r$ = all directed paths from ingredient nodes to final product in recipe *r*
- |p| = edge count (length) of path *p*
- |R| = total number of recipes in dataset

3.7 Ingredient Co-occurrence and Intermediate Product Comparison Methodology

This section describes methodologies for analyzing ingredient patterns across the dataset and comparison of intermediate product compositions for similarities

3.7.1 Ingredient Frequency Analysis

Ingredient frequency measures how commonly each ingredient appears across the recipe dataset. Unlike a simple ingredient count count, this analysis uses recipe-based frequency rather than occurrence-based counting. When the same ingredient appears multiple times in a single recipe (e.g.eggs used in both the crust and filling of a pie), it counts as one recipe occurrence to avoid improper counts; ingredients are deduplicated in order to receive only a single count of its occurrences in a recipe. We will calculate the frequency of ingredient *i* as follows:

$$frequency(i) = \frac{|\{r \in R : i \in ingredients\}|}{|R|} \cdot 100$$

Where:

- *R* is the set of all recipes
- ingredients(*r*) is the set of distinct ingredients in recipe *r*
- The numerator counts recipes containing ingredient *i* (regardless of how many times *i* appears within that recipe)

This deduplication ensures that rankings reflect true prevalence (how many recipes use the ingredient), rather than intensity (how often it appears within recipes)

3.7.2 Co-occurrence Analysis

Co-occurrence analysis identifies ingredient combinations (intermediate products) that appear together frequently across recipes. Our algorithm extracts ingredient sets from each recipe, generates all *N*-tuple combinations, and counts their frequency using metrics adapted from "market basket" analysis; finding frequent itemsets and their co-occurrence patterns. (Agrawal & Srikant, 1994)

Algorithm:

1. For each recipe r, extract the set of distinct ingredient names I_r
2. For a given tuple size N, generate all $\binom{|I_r|}{N}$ combinations
3. Count occurrences of each N-tuple across all recipes using a frequency counter
4. Calculate the support for each tuple: $support(t) = \frac{|\{r : t \subseteq I_r\}|}{|R|}$
5. Filter by minimum support threshold and sort by count (descending)

For pairs ($N = 2$), this reveals foundational ingredient relationships. For higher-order tuples ($N = 3$ to $N = 6$), this reveals recipe archetypes; ingredient clusters that define categories such as "basic cake batter" (flour + sugar + eggs + butter)

3.7.3 Intermediate Product Comparison

Intermediate products such as "batter", "dough", "filling", or "glaze" represent transformation stages within a recipe DAG. These are products produced by actions that serve as

inputs to subsequent actions, rather than the recipe's final product. The following analysis compares intermediate products across recipes to identify compositional similarity patterns. Intermediate products are identified in the DAG as any node that is produced by an action (in-edges from action nodes) and is consumed by subsequent actions (out-edges to another action). We are excluding final products which have no out-edges to subsequent actions.

To compare the overlap of ingredient sets we used a Jaccard Similarity metric to measure the proportion of shared ingredients between two intermediates. Jaccard Similarity is defined as the intersection divided by the union of two sets of ingredients. Below is an example of calculating Jaccard Similarity using two similar batters that we might encounter in the recipe dataset:

$$J(A,\ B)\ = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

- $0 \leq J(A,\ B) \leq 1$
- $J(A,\ B)\ = 1 \Leftrightarrow A\ = B$
- $J(A,\ B)\ = 0 \Leftrightarrow A\ \cap\ B\ = \emptyset$

Given:

Batter A = {flour, eggs, sugar, butter}

Batter B = {flour, eggs, milk}

$A\ \cap\ B\ =$ {flour, eggs} $\Rightarrow |A\ \cap\ B| = 2$

$A\ \cup\ B\ =$ {sugar, butter, flour, eggs, milk} $\Rightarrow\ |A\ \cup\ B| = 5$

$J(A,\ B)\ = \frac{2}{5} = 0.4$

Both batters share 40% of their ingredients

Cosine similarity measures how similar ingredients are in relation to their proportions, intermediate products will be treated as a vector of its constituent ingredients' quantities:

$$cos\left(\vec{A}, \vec{B}\right) = \frac{\vec{A} \cdot \vec{B}}{\left|\left|\vec{A}\right|\right| \times \left|\left|\vec{B}\right|\right|}$$

Where $\vec{A}$ and $\vec{B}$ are vectors of ingredient quantities (in standardized units). A cosine score of 1.0 indicates proportionally identical compositions, 0.0 indicates completely different proportions. Cosine similarity requires quantity information and is more sensitive to ratios than Jaccard; it makes distinctions such as "same ingredients, different amounts" or "double the butter" which Jaccard does not account for. Another approach we can take is performing cross-recipe comparison of intermediates that appear across multiple recipes. For example, "batter" appears in 5 recipes, "dough" appears in 6 recipes. We can calculate all pairwise comparisons across recipe boundaries ($5 \times 6 = 30$ comparisons) and report:

- Mean: Average similarity across all cross-recipe pairs
- Median: Middle value, robust against outliers
- Count: Number of recipe pairs compared

This aggregate approach will reveal whether intermediate products with the same name share consistent compositions across different recipes.

CHAPTER FOUR: RESULTS AND DISCUSSION

4.1 Directed Acyclic Graph Structure Analysis Results

Analysis of DAG structural metrics across our 76-recipe dataset revealed an average node count of 30.8 (range: 17-72), average edge count of 31.2 (range: 11-90), and an average critical path length of 17.1 edges (range: 4-48). The critical path represents the "deepest" sequence of transformations required of ingredients to achieve the final product. Higher critical path lengths indicate more recipe depth through transformative actions and therefore greater overall recipe complexity (Figure 4.1)

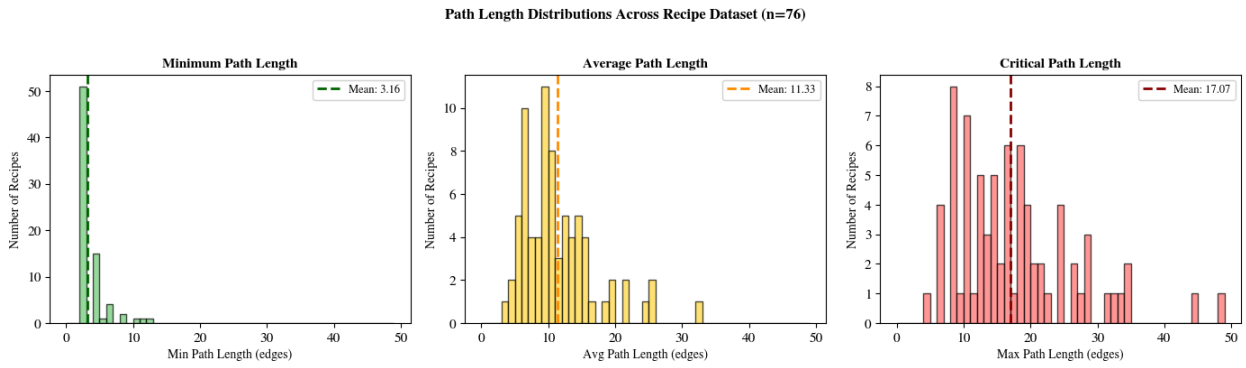


Figure 4.1 - Path Length Distributions Across Recipe Dataset (n=76)

Branch analysis revealed relatively low parallelism in our dataset, with an average branch count of only 1.07 (range: 1-2). Approximately 93% of recipes exhibited single-branch structures, meaning strictly sequential transformations. The other 7% of recipes contained two branches, representing parallel cooking processes. This low branch variance is characteristic of traditional baking recipes, where ingredient additions typically follow linear sequences rather than requiring simultaneous preparation. As we have focused primarily on baking recipes, most of the recipes with 2 branches usually are creating some kind of topping, icing, or glaze while the main branch product bakes, for example.

Within our dataset we have discovered three major insights:

1. High complexity variation: The 13.9 edge gap between average minimum path length (3.2 edges) and average critical path length (17.1 edges) shows that across all of our recipes, there is a wide variety of ingredient transformation chains. Some of the ingredients are simple additions while others undergo extensive transformation.
2. The mean is closer to the max: The average mean path length (11.3 edges) is closer to the average critical path length (17.1 edges) than to the average minimum path length (3.2 edges), suggesting most ingredients undergo substantial processing, with only a few "simple" ingredients
3. Wide range of paths: The edge range for the critical path length (4-48 edges) reveals that the dataset contains a mix of extremely simple recipes and highly complex multi-stage recipes.

In terms of the practical meaning of these results, a recipe with a critical path length of 48 edges, an average path length of 32.4 edges, and a minimum path length of 12 edges would be an extremely complex recipe in terms of ingredient count and transformations. While a recipe with a critical path length of 4 edges, an average path length of 3.5 edges, and a minimum path length of 2 edges would require far less ingredients and transformative actions.

4.2 Recipe Complexity Results

Applying our weighted complexity formula $C = 0.5B + 0.35A + 0.15I$ across the 76-recipe dataset (Appendix A) yielded complexity scores ranging from 2.30 to 9.85, with a mean of 4.58 and standard deviation of 1.72 (Figure 4.2).

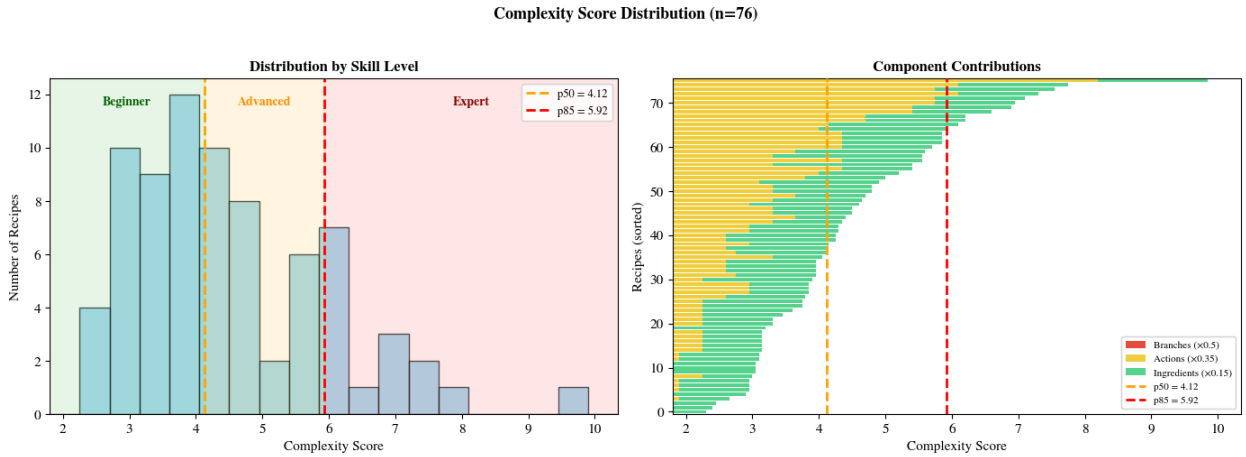


Figure 4.2 - Complexity Score Distribution (n=76)

Recipe	Branches (B)	Actions (A)	Ingredients (I)	Complexity (C)	Skill Level
Herb-Roasted Potatoes Poupon	1	3	7	2.60	Beginner
Sour Cream Coffee Cake	1	10	9	5.35	Advanced
Neopolitan Cookies	2	22	13	9.65	Expert

Table 3 - Recipe Complexity Score Examples

These examples demonstrate how our complexity formula weighs each recipe based on this criteria:

- Herb-Roasted Potatoes: $C = 0.5(1) + 0.35(3) + 0.15(7) = 0.5 + 1.05 + 1.05 = 2.60$
- Sour Cream Coffee Cake: $C = 0.5(1) + 0.35(10) + 0.15(9) = 0.5 + 3.5 + 1.35 = 5.35$
- Neapolitan Cookies: $C = 0.5(2) + 0.35(22) + 0.15(13) = 1.0 + 7.7 + 1.95 = 10.65$

The branch count contributes minimally given our dataset's low parallelism (93% single-branch), while action count serves as the primary factor for placement between skill levels. This aligns with what we might expect based on real-world cooking experience. Making Neopolitan Cookies from scratch is a two-day process with lots of intricate dough manipulation, layering, and cutting; we expected it to score the highest based on its known numerous transformative actions and concurrent steps. Recipes in the expert classification ($C \geq 5.93$) typically required 15+ actions, while beginner recipes averaged only 5 actions.

4.3 Ingredient Co-occurrence and Intermediate Product Comparison Results

Across 76 recipes we observed 203 unique ingredients over 815 occurrences, with 136 of those ingredients appearing in only a single recipe. This high specialization ratio (67% of ingredients appearing only once) reflects the diverse nature of our baking recipes, where many ingredients serve specific roles in individual recipes rather than appearing across multiple dishes.

4.3.1 Ingredient Frequency Results

Despite the high specialization, a small subset of ingredients dominate the dataset as presented in Table 4:

Ingredient	# of Recipes	% of Recipes
Sugar	49	64.5%
Flour	42	55.3%
Eggs	40	52.6%
Salt	37	48.7%
Butter	31	40.8%

Table 4 - Top 5 Most Common Ingredients in the Dataset

These five ingredients form the foundational components of baked goods: sugar for sweetness and structure with flour as the primary structural component. Eggs contribute to binding and leavening. Salt is necessary to enhance flavor as well as to control yeast activity. Butter is the main fat among these recipes, to add tenderness and flavor. Their high prevalence (appearing in ~41% - ~65% of our dataset) contrasts with the 141 single-use ingredients. This is evidence for a “long tail” distribution that might be indicative of culinary datasets, where a select few staple ingredients are present in the majority of recipes, as well as the combination of many specialized ingredients (Figure 4.3).

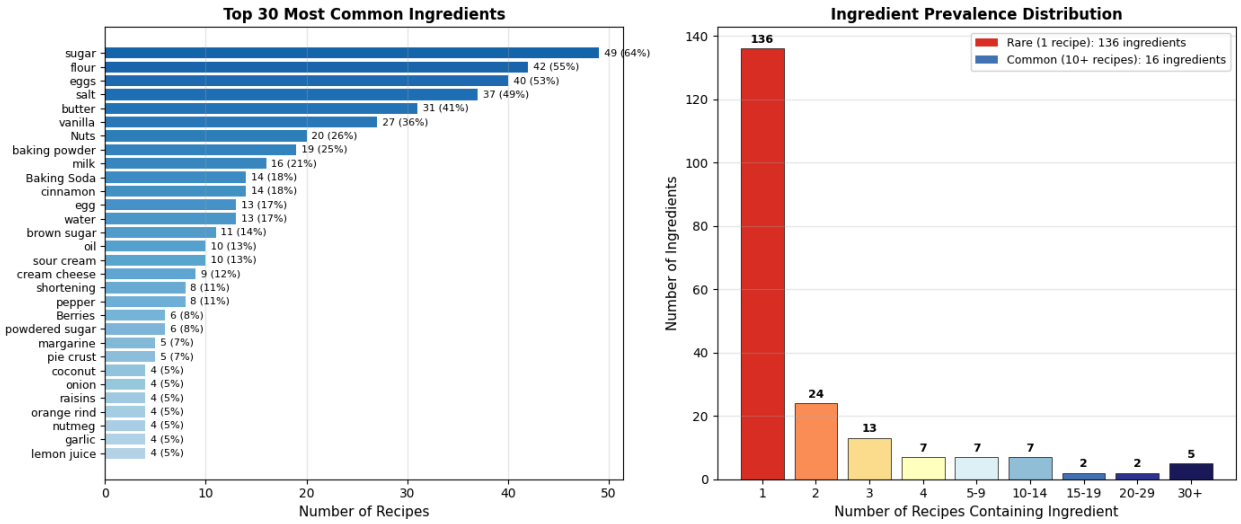


Figure 4.3 - Long-Tail Distribution of Ingredient Prevalence

4.3.2 Co-occurrence Analysis Results

The co-occurrence analysis identified ingredient pairs that frequently appear together across recipes, revealing fundamental baked goods composition: Table 5 presents the top five strongest ingredient pair associations. Table 6 presents the top five strongest ingredient triples.

Pair	# of Recipes	% of Recipes
Flour + Sugar	35	46.1%
Eggs + Sugar	32	42.1%
Flour + Salt	28	36.8%
Salt + Sugar	25	32.9%
Eggs + Flour	24	31.6%

Table 5 -Top 5 Strongest Ingredient Pair Associations

Triple	# of Recipes	% of Recipes	Top Combining Actions
Flour + Salt + Sugar	24	31.6%	combine (24%), mix (21%), sift (21%)
Eggs + Flour + Sugar	21	27.6%	mix (33%), add (20%), pour (13%)
Eggs + Sugar + Vanilla	21	27.6%	beat (29%), add (24%), mix (19%)
Butter + Flour + Sugar	19	25.0%	mix (26%), cream (26%), combine (21%)
Butter + Eggs + Sugar	18	23.7%	cream (29%), beat (18%), mix (18%)

Table 6 - Top 5 Strongest Ingredient Triple Associations

The pair analysis reveals that flour + sugar (46.1% prevalence) and eggs + sugar (42.1% prevalence) form the most fundamental combinations in our baking dataset. These pairs represent the core structural and sweetening components that define most naked goods. The flour + salt pair (36.8% prevalence) reflects salt's innate identities as a flavor enhancer and gluten developer in flour-based recipes.

The triple analysis provides even deeper insight into baking recipe archetypes. The flour + salt + sugar triple (31.6% prevalence) represents the essential dry ingredient base found in nearly one-third of our entire dataset. This combination forms the foundation of most cake batters, cookie doughs, and quick breads. The eggs + flour + sugar triple (27.6% prevalence) captures the complete structural foundation of cakes and enriched doughs, where eggs provide binding and leavening while flour and sugar contribute to the structure of the baked goods their recipes represent

Notably, the eggs + sugar + vanilla triple (27.6% prevalence) reveal a distinct preparation pattern: this combination typically appears in recipes where eggs and sugar are creamed or beaten together with vanilla flavoring before incorporating dry ingredients. This technique is common in pound cakes, custards, and many of our cookie recipes. The butter-containing triples (butter + flour + sugar at 25% prevalence and butter + eggs + sugar at 23.7% prevalence) identify recipes employing the creaming method, where butter is beaten with sugar to incorporate air in the mixture before eggs and flour (the structural/binding components) are added.

The progression from pairs to triples demonstrates how ingredient co-occurrence analysis can identify not just ingredient associations but also implied technique patterns. While pairs reveal what ingredients commonly appear together, triples begin to suggest how those ingredients are combined, distinguishing between: dry ingredient mixing, egg-sugar creaming, and butter-based preparations.

4.3.3 Intermediate Product Comparison Results

To investigate whether intermediate products with similar names exhibit similar compositions, we computed pairwise Jaccard similarity coefficients across all intermediate product instances appearing in two or more recipes. Our analysis identified 12 distinct intermediate product types occurring in at least two recipes, yielding 66 pairwise comparisons. Table 7 presents both the frequency of intermediate product types and the similarity scores for the most compositionally similar pairs.

Intermediate A	Intermediate B	Jaccard	Occurences	Comparisons
dry ingredients	dry mixture	0.286	(5, 6)	30
wet mixture	creamed mixture	0.216	(3, 2)	6
egg mixture	wet mixture	0.212	(3, 3)	9
batter	sifted dry ingredients	0.111	(5, 5)	25
batter	dry mixture	0.101	(5, 6)	30

Table 7 - Top 10 most Similar Intermediate Product Pairs

The most frequently occurring intermediate product is “dry mixture” (6 recipes), followed by “batter” and “sifted dry ingredients” (5 recipes each). Several intermediates appear in exactly 2 recipes, representing the minimum threshold for meaningful comparison.

The pairwise similarity analysis reveals that most intermediate pairs share little compositional overlap, with a mean Jaccard coefficient of 0.052. Only 36 of 66 pairs (54.5%) exhibit any shared ingredients. The most similar pair (“sifted dry ingredients” and “dry mixture”) achieves a Jaccard score of 0.286, reflecting their common use of both flour and leavening agents. The “wet mixture” and “creamed mixture” pair (0.216 Jaccard score) share butter and

sugar, while “egg mixture” and “wet mixture” (0.212 Jaccard score) commonly contain eggs and vanilla.

These results suggest that intermediate product naming conventions capture functional roles rather than strict compositional specifications. A “dry mixture” in one recipe may contain different ingredients than a “dry mixture” in another, depending on the specific dish being prepared. This finding has implications for recipe comparison systems: intermediate product types cannot be reliably substituted based on name alone, and successful composition estimation would require contextual analysis of the parent recipe

4.4 Discussion

According to the data above, we can see that the ingredient co-occurrence results suggest that the recipes in our dataset are highly specialized. Our recipes also show a large range and variability of complexity with some being quite simple and others considerably more complex in terms of inputs and transformation steps. Across 76 recipes we observed 203 unique ingredients over 815 occurrences, with 136 ingredients appearing in only a single recipe, corresponding to an average of 3.6 unique ingredients per recipe. We can also see that a small subset of ingredients dominate the dataset: sugar appears in 64.5% of recipes, flour in 55.3%, eggs in 52.6%, salt in 48.7% and butter in 40.8%. The co-occurrence analysis demonstrates ingredients that exist commonly together in many ingredients (e.g. egg + salt and flour + shortening).

The DAG structure analysis also indicates complexity: on average recipes contained 30.8 nodes, 31.2 edges, and a critical path length of 17.1 edges. These metrics revealed that recipes have mixed complexity where some ingredients have simple trajectories while in other recipes

they experience extensive processing, this can be seen from the difference in average minimum path length (3.2 edges) and average critical path length (17.1 edges). The edge range for the maximum paths (4 - 48 edges) again solidifies our result that there are very simplistic recipes contained in the dataset while others are multi-stage complex recipes. The DAGs were able to successfully capture the overall spectrum of complexity in recipes.

CHAPTER FIVE: CONCLUSIONS AND FUTURE WORK

5.1 Introduction

In this thesis we had two main research questions:

1. Is there an existing data structure that can allow for comparison of elements that contain a wide range of associated metadata?
2. What can quantitative metrics of DAGs such as maximal (critical) and average path lengths from ingredient nodes to the final product actually mean in the context of cooking?

Presently, we were able to address both research questions that we set out to investigate. Through the creation of DAGs that capture cooking semantics and an extensive toolset for decomposing inputs and comparing recipes, we can say that we have created a robust methodology for recipe analysis. Furthermore, we were able to quantify aspects of DAGs based on their innate structures such as topographical ordering and adjacency lists to understand the complexity of the dataset and the information provided to go from ingredients to intermediate products, resulting in a finished product. This process was not as time consuming as originally thought and demonstrates that recipes can be abstracted regardless of the language, semantics, and context of the original writer.

5.2. Conclusion of Research

Through our literature review we identified a knowledge gap in the ability to capture recipe semantics for practical complexity analysis. While prior work such as Donatelli et al. (2021) and Mori et al. (2014) constructed graph representations from recipe texts, these approaches focused primarily on linguistic alignment and flow representation rather than quantitative complexity metrics. None of the existing methods incorporate producer-consumer semantics, canonical symbol mapping, or practical software tools for recipe comparison and visualization. With our work we were able to represent recipes as DAGs in ways that both accounted for, and ignored cooking semantics to evaluate the complexity of them through anatomical DAG metrics (e.g. path lengths and intermediate ingredient composition).

Once our results were completed, we were able to successfully evaluate the complexity of our recipes and identify the most common ingredients. We found that our recipes ranged largely in complexity, demonstrating that most ingredients undergo drastic changes to produce the final result of a recipe. Although a few main ingredients dominated (e.g. sugar and flour) they were often paired with other ingredients as well. This is not an entirely surprising result, as we previously stated we were starting our methodology with baking recipes. However, it is interesting how much variation there is in only a small dataset, I had expected a more even distribution in complexity and difficulty. My main takeaways from this work is that it might be useful in problem categories other than recipes, in areas with processes that clearly define the transformation of inputs into one output whose identity is based on its constituent components. An example of this might be manufacturing, with inputs representing raw materials as well as actions and equipment representing in which space (or machine) and with what discrete methods that these raw materials transformed incrementally.

5.3. Future Work

The methodology and toolset developed in this thesis provide a foundation for future additions that would enhance both the analytical capabilities and practicality of the system. The current system requires manual JSON schema creation for recipe input. Future development could expand the ingestion capabilities of the dataset through multiple formats. Plain text parsing using natural language processing to extract ingredients, actions, and equipment from unstructured recipe text. Optical Character Recognition (OCR) could be used to digitize scanned handwritten recipe cards similar to those adapted for our dataset. Mapping of JSON schemas (online JSON recipe storage to our internal recipe JSON schema) would be a trivial addition. These new methods would dramatically reduce the barrier to populating the recipe database and enable analysis of larger, more diverse datasets.

Our current DAG representation treats all edges uniformly, but cooking actions vary significantly in duration and timing sensitivity. A natural extension of our DAG structure would be to assign edge weights that represent real-world action completion time. Since cooking times exhibit inherent variability in reality (e.g. baking chicken 45-60 minutes depending on thickness, temperature, altitude, etc.), user studies could collect empirical timing data across multiple executions of the same recipe. These observations would yield mean completion times with associated variance for each action type, providing a more realistic model of recipe execution. Aggregating timing data across recipes would enable calculation of expected total preparation time, identification of rate-limiting steps, and skill-based time predictions.

Many recipe aggregator websites offer limited functionality for matching available ingredients to possible recipes. Our alias symbol system could be used to represent a user's pantry. We could take the set intersection between pantry contents and recipe requirements, with

parameters for advanced search functions (by identity, name, cuisine type, etc.) This feature represents a natural extension of comparing our existing data structures to partial identity and symbol sets.

The methodology representing transformative processes as DAGs extend beyond culinary applications. Manufacturing workflows, chemical synthesis procedures, and assembly instructions share the same fundamental structure as inputs transformed through sequential and parallel operations into outputs. Applying our complexity metrics and comparison tools to these domains could validate the generalizability of our approach and reveal domain-specific adaptations required to apply our system in broader ways.

References

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), 487–499.

Algabli, S. (2020). *Learning the graph edit distance through embedding the graph matching* (Doctoral thesis). Department of Computer Engineering and Mathematics, Tarragona.

Allrecipes. (2024). About Us - Editorial Standards. https://www.allrecipes.com/about-us-6648102

Bryan, C. (2019, February 21). Stop shaming recipe bloggers for writing a lot. Mashable. https://mashable.com/article/recipe-blog-long-stories

Cormen, T.H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Demaine, E. D., Mozes, S., Rossman, B., & Weimann, O. (2006). An $O(n^3)$-time algorithm for tree edit distance. *arXiv preprint* cs/0604037.

Donatelli, L., Rippeth, E., Lai, K., & Koller, A. (2021). Aligning Actions Across Recipe Graphs. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 6930–6942. https://doi.org/10.18653/v1/2021.emnlp-main.554

Eder, J., & Wiggisser, K. (2006). A DAG comparison algorithm and its application to temporal data warehousing. In *Lecture Notes in Computer Science* (Conference paper). https://doi.org/10.1007/11908883_27

Gao, X., Xiao, B., Tao, D., & Li, X. (2010). A survey of graph edit distance. *Pattern Analysis and Applications, 13*, 113–129. https://doi.org/10.1007/s10044-008-0141-y

Lin, I.-J., & Kung, S. Y. (n.d.). Coding and comparison of DAGs as a novel neural structure with applications to on-line handwriting recognition. Unpublished manuscript, Princeton University.

Malmi, E., Tatti, N., & Gionis, A. (n.d.). Beyond rankings: comparing directed acyclic graphs. Manuscript submitted for publication.

Mori, S., Maeta, H., Yamakata, Y., & Sasada, T. (2014). Flow Graph Corpus from Recipe Texts. Proceedings of the 9th International Conference on Language Resources and Evaluation (LREC 2014), 2370–2377.

Nayak, G., Dutta, S., Ajwani, D., Nicholson, P., & Sala, A. (n.d.). Automated assessment of knowledge hierarchy evolution: Comparing directed acyclic graphs. Manuscript.

Schema.org. (2024). *Recipe Schema Specification*. https://schema.org/Recipe

U.S. Copyright Office. (2023). *Copyright Registration for Recipes*. Circular 33. https://www.copyright.gov/circs/circ33.pdf

Walpole, R. E., Myers, R. H., Myers, S. L., & Ye, K. (2012). Probability and Statistics for Engineers and Scientists (9th ed.). Pearson.

Appendices

Appendix A: Recipe Dataset

Recipe	Ingredients	Equipment	Actions	Difficulty Score
7 Up Cake	9	1	6	4.10
Apple Cake	10	1	3	3.05
Apple Coffee Cake	12	1	6	4.90
Apple Crisp	8	1	5	3.95
Artichoke Bread	7	1	4	2.95
Banana Split Pie	11	3	16	7.75
Banana Walnut Bread	9	2	6	3.95
Barbequed Country-Style Spareribs	9	3	8	4.65
Batter Buns	7	3	9	4.70
Beef Taco Bake	6	1	7	3.85
Broccoli Casserole	8	1	7	4.15
Buttermilk Pie	6	0	5	3.15
Candied Yams	5	1	4	2.65
Caramel Pecan Rolls	8	3	15	6.95
Carrot Cake	11	1	3	3.20
Cheesecake	6	1	5	3.15
Cheesecake	6	1	5	3.15

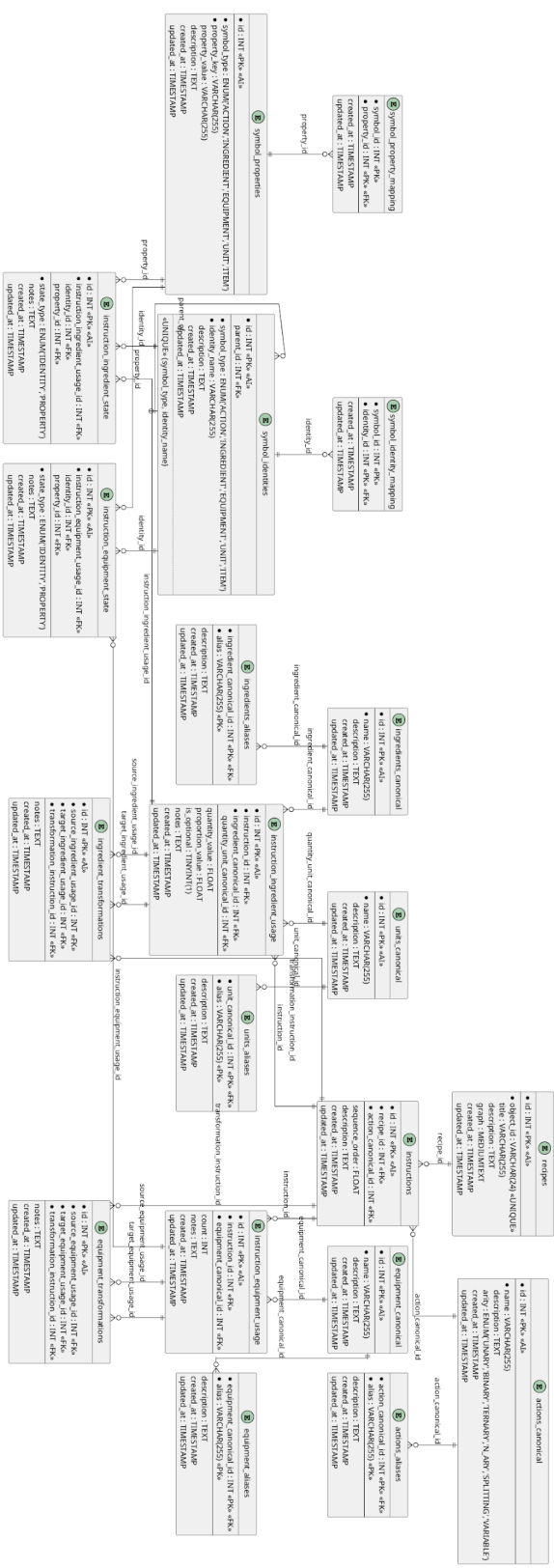
Recipe	Ingredients	Equipment	Actions	Difficulty Score
Cherry Pie Supreme	7	1	7	3.85
Chocolate Chip Muffins	10	4	6	4.10
Coke Cake	11	4	6	4.25
Corn Pudding	6	2	5	3.15
Cornbread	8	2	4	3.10
Cranberry Nut Bread	10	4	11	5.85
Cream Cheese Brownie Pie	9	5	15	7.10
Crunch-Topped French Toast	10	5	12	6.20
Date-Nut Bread	10	1	3	3.05
Double Chocolate Oatmeal Cookies	11	2	6	4.25
Double Layer Pumpkin Pie	10	3	8	4.80
Easy Focaccia Bread	12	5	15	7.55
Easy Refrigerator Rolls	7	8	11	5.40
Eggplant Casserole	8	4	16	7.30
Farina Pie	14	1	9	5.60
Fruit Cake	7	1	4	2.95
Gold Loaf Cake	9	1	7	4.30
Graham Streusel	9	2	5	3.60

Recipe	Ingredients	Equipment	Actions	Difficulty Score
Cake				
Herb-Roasted Potatoes Poupon	5	1	3	2.30
Italian Cheesecake	10	2	5	3.75
Italian Cream Tarts	10	2	14	6.90
Lemon Bars	7	1	8	4.35
Mini Cheesecakes	5	2	8	4.40
Mini-Orange Crumble Top Muffins with Orange Butter	13	2	10	5.95
Mississippi Mud Cake	11	2	7	4.60
Neopolitan Cookies	11	6	22	9.85
New York Luncheonette Corn Muffins	8	4	8	4.50
No Crust Pie	8	3	2	2.40
Noodle Pudding	9	2	3	2.90
Oatmeal Cookies	10	3	5	3.75
One-Dish Chicken and Rice	6	3	5	3.15
Pear Berry Pie	10	3	3	3.05
Pecan Pie	6	3	3	2.45
Pine Nut Macaroons	5	6	8	4.05

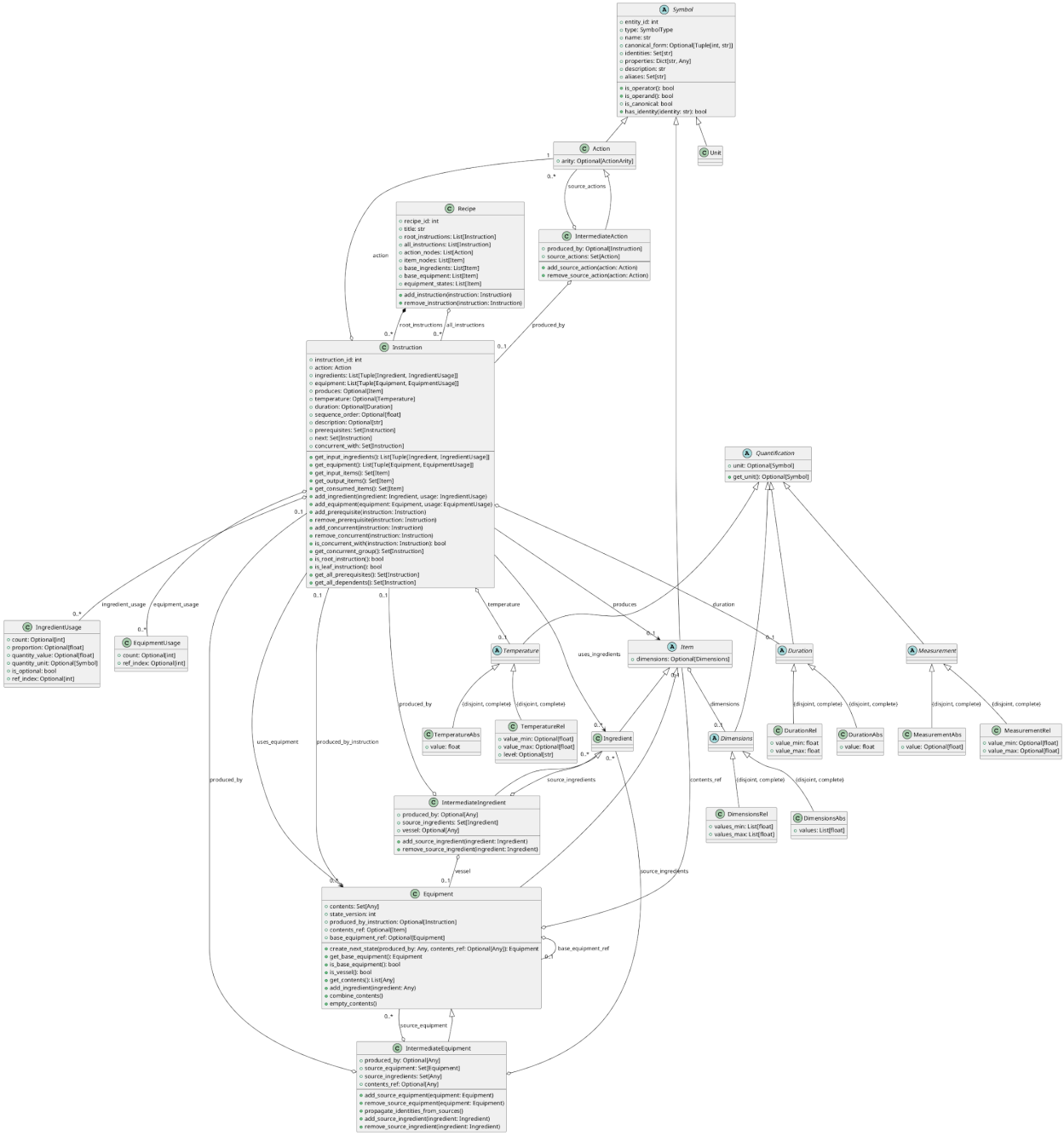
Recipe	Ingredients	Equipment	Actions	Difficulty Score
Pineapple Cheese Dish	6	1	7	3.85
Poppy Farm Cake	10	2	11	5.85
Pork Tenderloin	8	3	10	5.20
Pot Cheese Cookies	7	2	5	3.30
Potatoes Au Gratin	7	4	5	3.30
Pumpkin Cookies	9	2	6	3.95
Pumpkin Muffins	11	2	5	3.90
Pumpkin Praline Dessert	8	2	5	3.45
Quick and Cheesy Chicken Enchiladas	8	3	9	4.50
Raw Apple Cake	15	2	8	5.55
Roasted Potatoes and Green Beans	10	2	12	6.20
Salmon Wellington	5	2	5	3.00
Scones	8	5	11	5.55
Secret Recipe Chocolate Chip Cookies	13	5	9	6.10
Sour Cream Coffee Cake	10	2	8	4.80
Sparkling Macaroon Chews	9	2	7	4.30

Recipe	Ingredients	Equipment	Actions	Difficulty Score
Spinach & Bacon Quiche	8	1	4	3.10
Strawberry-Rhubarb Pie	8	4	14	6.60
Sweet Potato-Pineapple Bake	8	5	8	5.00
Swiss-Stuffed Green Peppers	9	3	11	5.70
Toll House Cookies	10	4	11	5.85
Tomato Pie	8	1	6	3.80
Victory Sheet Cake	7	2	4	2.95
Vidalia Onion Pie	9	1	6	3.95
Zucchini Bread	14	4	8	5.40

Appendix B: Relational Database Structure



Appendix C: Hierarchical Class Structure



Appendix D: DAG Building Algorithm and Helper Algorithms

Build DAG from Recipes (S, I):

Input:

- Recipe Steps $S_1, S_2, \ldots, S_n$
- Ingredients $I_1, I_2, \ldots, I_n$

Output:

- Directed acyclic graph G(V, E)

Steps:

1. V = ∅, E = ∅,
   EquipmentStates = {},
   IngredientStates = {},
   VesselContents = {},
   StartLevels = {},
   CompletionLevels = {},
   ProducerConsumerMap = {}
2. for each ingredient Ii ∈ I:
   2.1. IngredientStates[Ii.id] = Ii
   2.2. Ii.proportion = 1.0
   2.3. Identify explicitly consumed intermediate items:
      2.3.1. ConsumedIntermediates = ∅
      2.3.2. for each instruction Inst ∈ S:
         2.3.2.1. for each ingredient Ing in Inst.input_ingredients:
            2.3.2.1.1. if Ing is IntermediateIngredient:
               2.3.2.1.1.1. ConsumedIntermediates = ConsumedIntermediates ∪ {Ing.name}
         2.3.2.2. for each equipment Eq in Inst.equipment:
            2.3.2.2.1. if Eq is IntermediateEquipment:
               2.3.2.2.1.1. ConsumedIntermediates = ConsumedIntermediates ∪ {Eq.name}
   2.4. Assign branch identifiers:
      2.4.1. BranchMap = {}
      2.4.2. branch_counter = 0
      2.4.3. roots = {Inst ∈ S : Inst.prerequisites = ∅}
      2.4.4. for each Root in roots:
         2.4.4.1. BranchMap[Root.id] = branch_counter
         2.4.4.2. branch_counter++
      2.4.5. for each instruction Inst in topological order:
         2.4.5.1. if Inst.id ∉ BranchMap:

2.4.5.1.1. prerequisite_branches = {BranchMap[P.id] : P ∈ Inst.prerequisites}
2.4.5.1.2. if |prerequisite_branches| = 1:
2.4.5.1.2.1. BranchMap[Inst.id] = prerequisite_branches[0]
2.4.5.1.3. else if |prerequisite_branches| > 1:
2.4.5.1.3.1. BranchMap[Inst.id] = min(prerequisite_branches)
2.5. Compute instruction levels (supporting sequential multi-root, separate branches, and duration):
2.5.1. Initialize start and completion levels for root instructions:
2.5.1.1. Collect all root instructions (no prerequisites)
2.5.1.2. Group roots by concurrency:
2.5.1.2.1. concurrent_groups = []
2.5.1.2.2. sequential_roots = []
2.5.1.2.3. for each root instruction Root:
2.5.1.2.3.1. if Root has 'concurrentWith' field:
2.5.1.2.3.1.1. find or create group with Root's concurrent partners
2.5.1.2.3.1.2. add Root to that concurrent_group
2.5.1.2.3.2. else:
2.5.1.2.3.2.1. sequential_roots.append(Root)
2.5.1.3. Assign levels to roots:
2.5.1.3.1. current_level = 1
2.5.1.3.2. for each concurrent_group in concurrent_groups:
2.5.1.3.2.1. for each Root in concurrent_group:
2.5.1.3.2.1.1. StartLevels[Root.id] = current_level
2.5.1.3.2.1.2. CompletionLevels[Root.id] = current_level
2.5.1.3.2.2. current_level += 2
2.5.1.3.3. for each Root in sequential_roots:
2.5.1.3.3.1. StartLevels[Root.id] = current_level
2.5.1.3.3.2. CompletionLevels[Root.id] = current_level
2.5.1.3.3.3. if Root produces output:
2.5.1.3.3.3.1. current_level += 2
2.5.1.3.3.4. else:
2.5.1.3.3.4.1. current_level += 1
2.5.2. Compute levels for remaining instructions (within-branch dependencies):
2.5.2.1. First pass - compute natural levels based on prerequisites:
2.5.2.1.1. changed = true
2.5.2.1.2. while changed:
2.5.2.1.2.1. changed = false
2.5.2.1.2.2. for each instruction Inst in topological order:
2.5.2.1.2.2.1. if Inst.id in StartLevels:
2.5.2.1.2.2.1.1. continue

2.5.2.1.2.2.2. if Inst has no prerequisites:
2.5.2.1.2.2.2.1. continue
2.5.2.1.2.2.3. if all prerequisites have been processed:
2.5.2.1.2.2.3.1. max_level = 0
2.5.2.1.2.2.3.2. for each prerequisite Prereq in Inst.prerequisites:
2.5.2.1.2.2.3.2.1. consumes_output = false
2.5.2.1.2.2.3.2.2. if Prereq produces output O:
2.5.2.1.2.2.3.2.2.1. if O in Inst's input ingredients:
2.5.2.1.2.2.3.2.2.1.1. consumes_output = true
2.5.2.1.2.2.3.2.2.2. if O in Inst's equipment:
2.5.2.1.2.2.3.2.2.2.1. consumes_output = true
2.5.2.1.2.2.3.2.3. if consumes_output:
2.5.2.1.2.2.3.2.3.1. level = CompletionLevels[Prereq.id] + 2
2.5.2.1.2.2.3.2.4. else:
2.5.2.1.2.2.3.2.4.1. if Prereq produces output:
2.5.2.1.2.2.3.2.4.1.1. level = CompletionLevels[Prereq.id] + 2
2.5.2.1.2.2.3.2.4.2. else:
2.5.2.1.2.2.3.2.4.2.1. level = CompletionLevels[Prereq.id] + 1
2.5.2.1.2.2.3.2.5. max_level = max(max_level, level)
2.5.2.1.2.2.3.3. StartLevels[Inst.id] = max_level
2.5.2.1.2.2.3.4. CompletionLevels[Inst.id] = max_level
2.5.2.1.2.2.3.5. changed = true
2.5.2.2. Second pass - adjust concurrent instructions to share max level:
2.5.2.2.1. concurrent_groups = []
2.5.2.2.2. processed = set()
2.5.2.2.3. for each instruction Inst ∈ S:
2.5.2.2.3.1. if Inst in processed: continue
2.5.2.2.3.2. if Inst has no 'concurrentWith' field: continue
2.5.2.2.3.3. group = get_concurrent_group(Inst)
2.5.2.2.3.4. if len(group) > 1:
2.5.2.2.3.4.1. concurrent_groups.append(group)
2.5.2.2.3.4.2. processed.update(group)
2.5.2.2.4. for each group in concurrent_groups:
2.5.2.2.4.1. max_group_level = max(StartLevels[Inst.id] for Inst in group)
2.5.2.2.4.2. for each Inst in group:
2.5.2.2.4.2.1. StartLevels[Inst.id] = max_group_level

2.5.2.2.4.2.2. CompletionLevels[Inst.id] = max_group_level
2.5.3. Enforce cross-branch sequential execution (third pass):
2.5.3.1. Check for level conflicts:
2.5.3.1.1. for each level L from 1 to max_level:
2.5.3.1.1.1. instructions_at_level = [Inst where StartLevels[Inst.id] == L]
2.5.3.1.1.2. if len(instructions_at_level) > 1:
2.5.3.1.1.2.1. check if all instructions are marked concurrent
2.5.3.1.1.2.2. if NOT all concurrent AND from different branches:
2.5.3.1.1.2.2.1. keep first instruction at level L
2.5.3.1.1.2.2.2. shift remaining instructions down by 2 levels
2.5.3.1.1.2.2.3. recursively update all dependent instructions
2.5.4. Adjust completion levels for duration-based actions (fourth pass):
2.5.4.1. for each instruction Inst ∈ S:
2.5.4.1.1. if Inst has duration AND Inst produces output O:
2.5.4.1.1.1. consumers = ProducerConsumerMap[Inst.id].consumers
2.5.4.1.1.2. if consumers is not empty:
2.5.4.1.1.2.1. consumer_levels = [StartLevels[C.id] for C in consumers]
2.5.4.1.1.2.2. earliest_consumer = min(consumer_levels)
2.5.4.1.1.2.3. CompletionLevels[Inst.id] = earliest_consumer - 2
(Output appears at earliest_consumer - 1, so completion = earliest_consumer - 2 since output_level = CompletionLevel + 1)
3. DAG Construction - Create nodes V and edges E:
3.1. Initialize node creation phase:
3.1.1. V = ∅ (vertex set)
3.1.2. E = ∅ (edge set)
3.1.3. IngredientNodes = {} (maps NodeId to Ingredient object)
3.1.4. EquipmentNodes = {} (maps NodeId to Equipment object)
3.1.5. InstructionIngredientMap = {} (maps (inst_id, refIndex) to NodeId)
3.1.6. InstructionEquipmentMap = {} (maps (inst_id, refIndex) to NodeId)
3.1.7. VesselContents = {} (tracks what's in each vessel)
3.2. Create ingredient and equipment input nodes:
3.2.1. for each instruction Inst ∈ S:
3.2.1.1. start_level = StartLevels[Inst.id]
3.2.1.2. for each ingredient Ing in Inst.input_ingredients:
3.2.1.2.1. Create unique node for this usage:
3.2.1.2.1.1. NodeId = f'{Ing.name}_inst{Inst.id}_ref{Ing.refIndex}'
3.2.1.2.1.2. V = V ∪ {NodeId} with level(NodeId) = start_level - 1
3.2.1.2.1.3. IngredientNodes[NodeId] = Ing
3.2.1.2.1.4. InstructionIngredientMap[(Inst.id, Ing.refIndex)] = NodeId

3.2.1.3. for each equipment Eq in Inst.equipment:
3.2.1.3.1. Create unique node for this usage:
3.2.1.3.1.1. NodeId = f'{Eq.name}_inst{Inst.id}_ref{Eq.refIndex}'
3.2.1.3.1.2. V = V ∪ {NodeId} with level(NodeId) = start_level - 1
3.2.1.3.1.3. EquipmentNodes[NodeId] = Eq
3.2.1.3.1.4. InstructionEquipmentMap[(Inst.id, Eq.refIndex)] = NodeId
3.3. Create prerequisite edges (structural dependencies):
3.3.1. for each instruction Inst ∈ S:
3.3.1.1. for each prerequisite Prereq ∈ Inst.prerequisites:
3.3.1.2. Skip edge if output consumed:
3.3.1.2.1. if Prereq produces output O:
3.3.1.2.1.1. if O ∈ Inst.input_ingredients OR O ∈ Inst.equipment:
3.3.1.2.1.1.1. continue (skip - handled by consumption edge)
3.3.1.3. Skip edge if branches converge:
3.3.1.3.1. if Prereq produces output:
3.3.1.3.1.1. if BranchMap[Prereq.id] ≠ BranchMap[Inst.id]:
3.3.1.3.1.1.1. continue (skip - cross-branch convergence)
3.3.1.4. Create sequential prerequisite edge:
3.3.1.4.1. E = E ∪ {(Prereq.action, Inst.action)}
3.4. Create consumption edges and update state:
3.4.1. for each instruction Inst ∈ S:
3.4.1.1. start_level = StartLevels[Inst.id]
3.4.1.2. V = V ∪ {Inst.action} with level(Inst.action) = start_level
3.4.1.3. for each equipment Eq in Inst.equipment:
3.4.1.3.1. NodeId = InstructionEquipmentMap[(Inst.id, Eq.refIndex)]
3.4.1.3.2. E = E ∪ {(NodeId, Inst.action)}
3.4.1.4. for each ingredient Ing in Inst.input_ingredients:
3.4.1.4.1. NodeId = InstructionIngredientMap[(Inst.id, Ing.refIndex)]
3.4.1.4.2. E = E ∪ {(NodeId, Inst.action)}
3.5. Create output nodes and production edges:
3.5.1. for each instruction Inst ∈ S:
3.5.1.1. completion_level = CompletionLevels[Inst.id]
3.5.1.2. if Inst transforms equipment:
3.5.1.2.1. for each equipment Eq transformed:
3.5.1.2.1.1. TransformedEq = ProcessEquipment(Eq, Inst)
3.5.1.2.1.2. V = V ∪ {TransformedEq} with level(TransformedEq) = completion_level + 1
3.5.1.2.1.3. E = E ∪ {(Inst.action, TransformedEq)}

3.5.1.2.1.4. Transfer vessel contents:
3.5.1.2.1.4.1. if VesselContents[Eq.id] exists:
3.5.1.2.1.4.1.1. VesselContents[TransformedEq.id] = VesselContents[Eq.id]
3.5.1.3. else if Inst produces output O:
3.5.1.3.1. Apply output filtering rules:
3.5.1.3.1.1. is_consumed = (O ∈ ConsumedIntermediates)
3.5.1.3.1.2. is_final_product = (O is recipe final product)
3.5.1.3.1.3. is_vessel_contents = false
3.5.1.3.1.4. Extract vessel contents if present:
3.5.1.3.1.4.1. if O matches pattern "container_name [contents: content_name]":
3.5.1.3.1.4.1.1. is_vessel_contents = true
3.5.1.3.1.4.1.2. container_name = extract container
3.5.1.3.1.4.1.3. content_name = extract contents
3.5.1.3.1.5. if is_consumed OR is_final_product:
3.5.1.3.1.5.1. if NOT is_vessel_contents:
3.5.1.3.1.5.1.1. Create output node
3.5.1.3.2. Create output node and edges:
3.5.1.3.2.1. InputIngredients = [Ing for Ing in Inst.input_ingredients]
3.5.1.3.2.2. InputEquipment = [Eq for Eq in Inst.equipment]
3.5.1.3.2.3. if |InputIngredients| > 0:
3.5.1.3.2.3.1. TransformedIng = ProcessIngredients(InputIngredients, Inst)
3.5.1.3.2.3.2. V = V ∪ {TransformedIng} with level(TransformedIng) = completion_level + 1
3.5.1.3.2.3.3. E = E ∪ {(Inst.action, TransformedIng)}
3.5.1.3.2.3.4. for each Ing ∈ InputIngredients:
3.5.1.3.2.3.4.1. IngredientStates[Ing.name] = TransformedIng
3.5.1.3.2.3.5. Handle vessel with contents:
3.5.1.3.2.3.5.1. if vessel V ∈ InputEquipment AND NOT is_vessel_contents:
3.5.1.3.2.3.5.1.1. FilledVessel = HandleVesselContents(V, InputIngredients, Inst)
3.5.1.3.2.3.5.1.2. V = V ∪ {FilledVessel} with level(FilledVessel) = completion_level + 1
3.5.1.3.2.3.5.1.3. E = E ∪ {(Inst.action, FilledVessel)}
3.5.1.3.2.3.5.1.4. VesselContents[FilledVessel.id] = InputIngredients
3.5.1.3.2.3.5.1.5. EquipmentStates[V.name] = FilledVessel
3.5.1.3.2.4. else:

3.5.1.3.2.4.1. Create generic result R
3.5.1.3.2.4.2. V = V ∪ {R} with level(R) = completion_level + 1
3.5.1.3.2.4.3. E = E ∪ {(Inst.action, R)}

4. return G(V, E)

NormalizeAndCompare(measurement1, measurement2) → proportion

- Input: Two measurements with potentially different units
- Output: The proportion of measurement1 relative to measurement2
- Purpose: Converts units and determines what fraction is being used

CombineActions(actions) → intermediateAction

- Input: Array of 2+ action objects
- Output: A new intermediate action with combined properties
- Purpose: Handles concurrent actions by combining them into a single action node

CombineIngredients(ingredients) → intermediateIngredient

- Input: Array of 2+ ingredient objects
- Output: A new intermediate ingredient with combined properties
- Purpose: Handles pure combination without transformation

ProcessIngredients(ingredients, action) → intermediateIngredient

- Input: Array of ingredient objects and an action object
- Output: A transformed intermediate ingredient
- Purpose: Transforms ingredients based on action type

ProcessEquipment(equipment, action) → intermediateEquipment

- Input: Equipment object and an action object
- Output: A transformed equipment object
- Purpose: Transforms equipment based on action

HandleVesselContents(vessel, ingredients, action) → filledVessel

- Input: Vessel equipment, ingredient array, and action
- Output: Vessel with contents information updated
- Purpose: Manages adding/removing contents from vessels